HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Entwurf und Implementierung eines Instanziierungsservices für variable Programmieraufgaben

Robin Drangmeister

Bachelor-Arbeit im Studiengang „Angewandte Informatik“ an der Fakultät IV – Wirtschaft und Informatik Hochschule Hannover

25. Juni 2018

Autor Robin Drangmeister
Matrikelnummer: 1381287
robin.drangmeister@outlook.com

Erstprüfer: Prof. Dr. Robert Garmann
Abteilung Informatik, Fakultät IV
Hochschule Hannover
robert.garmann@hs-hannover.de

Zweitprüfer: Peter Fricke, B. Sc.
E-Learning Center, Zentrum für Lehre und Beratung
Hochschule Hannover
peter.fricke@hs-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 25. Juni 2018 Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Zu einer Vielzahl von Vorlesungen gibt es Übungsaufgaben anhand derer das theoretisch erlernte Wissen noch weiter vertieft, gefestigt oder praktisch angewandt werden kann. Klassischerweise erstellt die Lehrperson hierfür eine Reihe von Aufgaben, die an den aktuellen Wissensstand und das bereits Gelernte angepasst sind. Alle Studierenden erhalten dann dieselben Aufgaben und je nach Modus können sie entweder freiwillig oder müssen verpflichtend bearbeitet werden.

Wenngleich sich dieses Konzept über viele Jahre bewährt hat, so gibt es doch eine Reihe von guten Gründen, warum nicht alle Studierenden dieselben Aufgaben bekommen sollten. Einige dieser Gründe werden in [3] herausgearbeitet. So sei es durchaus möglich, dass Studierende einen unterschiedlichen Lernstand besitzen und deswegen Aufgaben mit unterschiedlichem Schwierigkeitsgrad verteilt werden könnten. Auch sei es sinnvoll mehrere Aufgaben desselben Typs mehrfach hintereinander zu üben, um ein bestimmtes Konzept zu verinnerlichen. Als weiterer Punkt wird genannt, dass häufig auch Lösungen von anderen Studierenden genommen, abgegeben und somit als eigene Lösungen ausgegeben werden. Ebenfalls wird angeführt, dass bei „normalen“ Aufgaben die Problematik besteht, dass Lösungen von Jahr zu Jahr weitergereicht werden können.

Daraus ergibt sich, dass individuelle Aufgaben diverse Vorteile gegenüber „klassischen“ Aufgaben haben und das Lernen hierdurch noch weiter gefördert wird. Allerdings ist es nahezu unmöglich für die große Anzahl an Studierenden, die eine Vorlesung besuchen, pro Person eine individuelle Aufgabe zu erstellen, die an ihren Lernstand angepasst ist, dennoch nicht zu leicht oder zu schwer im Vergleich zu den anderen Aufgaben ist und die von der Thematik zum Thema und den anderen Aufgaben passt. Gewünscht wäre also ein Konzept, dass es erlaubt, mit einem ähnlichen Aufwand wie beim Erstellen von „normalen“ Aufgaben, auch individuelle Aufgaben zu erstellen.

In [3] wird ein Vorschlag gemacht, wie klassische Aufgaben in Aufgabenschablonen konvertiert werden bzw. diese Aufgabenschablonen direkt anstelle von normalen Aufgaben erstellt werden können. Aus diesen Aufgabenschablonen können dann automatisiert Aufgabeninstanzen generiert werden, die die Studierenden als normale Übungsaufgabe erhalten.

Diese Bachelorarbeit wird sich mit dem Entwurf und der Implementierung eines Services beschäftigen, der in der Lage sein soll, aus einer Aufgabenschablone eine Aufgabeninstanz zu generieren und diese zurückzugeben. Der Service soll in die in Abschnitt 2.3 beschriebene Systemlandschaft integriert werden, allerdings nicht nur dort speziell

laufen, sondern allgemeine Schnittstellen benutzen, sodass auch andere Systeme ähnlichen Aufbaus unterstützt werden.

Zunächst wird in Kapitel 2 die Ausgangssituation beschrieben, ebenso wie die bekannten Konzepte, Systeme und Formate, auf denen diese Arbeit aufbaut. Anschließend wird in Kapitel 3 der Zustand beschrieben, der mithilfe der in dieser Arbeit zu erstellenden Komponenten erreicht werden soll. Hierfür werden sowohl die Abläufe aus Nutzer-Sicht geschildert, als auch erläutert, wie die beteiligten Komponenten intern zusammenarbeiten. Darauf folgend wird in Kapitel 4 die bei der Erstellung dieser Arbeit und der Entwicklung der Komponenten verwendete Methodik vorgestellt. In Kapitel 5 werden unterschiedliche Lösungsansätze für Teile des Instanziierungsservices diskutiert. Insbesondere, welche Aufgaben im Einzelnen dem Instanziierungsservice selbst und welche einem zusätzlichen externen Backend zufallen könnten, sowie Möglichkeiten, diesen beiden Komponenten mitzuteilen, welche Teile einer Aufgabenschablone wie instanziiert werden sollen. Anschließend wird in Kapitel 6 der letztlich gewählte Entwurf vorgestellt und auf einige Einzelheiten der Implementierung eingegangen, bevor in Kapitel 7 die erstellten Komponenten bezüglich der gestellten Anforderungen überprüft und bewertet werden. Kapitel 8 fasst die Arbeit noch einmal kurz zusammen und Kapitel 9 bietet einen Ausblick auf mögliche Erweiterungen und Verbesserungsvorschläge der erstellten Komponenten.

2 Ist-Zustand

Bei den Aufgaben, die in der Einleitung bereits genannt wurden, handelt es sich um automatisiert-bewertete Programmieraufgaben. Der Unterschied zu normalen Aufgaben liegt in der Art der Bewertung: Bei manuell bewerteten Aufgaben werden die Abgaben der Studierenden von der Lehrperson nach bestimmten Kriterien bewertet, die dann entscheidend für die erreichte Punktzahl sind. Bei automatisiert-bewerteten Aufgaben erfolgt die Bewertung zweistufig. Zunächst erfolgt eine elektronische Abgabe der Arbeitsergebnisse, die dann an einen Grader weitergereicht und von diesem auf vorgegebene Aspekte untersucht und bewertet werden. Wenngleich die automatische Bewertung häufig sehr gute und korrekte Ergebnisse liefert, so kann es dennoch zu Fehlern kommen, weshalb sie anschließend noch einmal manuell von der Lehrperson überprüft, ggf. korrigiert oder mit zusätzlichem Feedback ergänzt wird. Bei den im 1. Kapitel genannten Aufgaben (den klassichen, variablen, den Aufgabenschablonen und auch den Aufgabeninstanzen) handelt es sich jeweils um automatisiert-bewertete Aufgaben. Falls nicht explizit anders genannt, sind mit Aufgaben in dieser Arbeit stets automatisiert-bewertete gemeint. Im Weiteren Verlauf dieses Kapitels wird nun beschrieben, auf welche Weise solche Aufgaben an der Hochschule Hannover zum Einsatz kommen, wie die Systemumgebung aussieht und somit gleichzeitig auch, welches die Nachbarsysteme des in dieser Arbeit zu erstellenden Services sind.

2.1 ProFormA-Austauschformat

Der folgende Abschnitt basiert auf der Ausarbeitung in [6]. Hier finden sich auch noch tiefer gehende Details zu den folgend genannten Systemen und Konzepten.

Da das Erstellen von (guten) Übungsaufgaben mitunter viel Zeit in Anspruch nehmen kann, wäre es wünschenswert, mit vielen Personen zusammen einen Aufgaben-Pool aufzubauen, der allen Beteiligten zur freien Verfügung steht. Pro Lehrperson ist es bereits üblich, dass diese mit der Zeit eine Reihe von Aufgaben erstellt, die dann jahrgangsübergreifend verwendet werden können, sodass häufig schon ein privater Pool an Aufgaben besteht. Dieser kann unter Umständen auch noch mit Kollegen geteilt werden, allerdings kann es bereits zu Problemen führen, falls er an andere Fakultäten derselben Hochschule weitergegeben wird. Nämlich genau dann, wenn dort bereits ein anderes System zur (automatisierten) Bewertung etabliert ist. Spätestens aber wenn der Austausch mit anderen Hochschulen erfolgen soll, wird es (ohne entsprechende

Anpassungen) sehr wahrscheinlich zu Problemen kommen, da die meisten Hochschulen eigens entwickelte Systeme mit unterschiedlichen Eigenschaften, Stärken und Schwächen verwenden, die häufig auf dort bereits bestehende Systeme aufgesetzt sind.

Aus diesem Grund wurde im Rahmen des eCult-Projektes[1] durch die ProFormA-Gruppe[2] ein XML-basiertes Austauschformat[3] für Programmieraufgaben entwickelt. Mithilfe dieses Formats ist es möglich, standardisierte Aufgaben zu erstellen, die von allen Systemen, die dieses Format unterstützen, verstanden und verarbeitet werden können.

Im ProFormA-Format werden Aufgaben als „task“ bezeichnet. Jede task enthält eine Beschreibung der Aufgabe, die Programmiersprache, mitgelieferte Dateien, technische Details zur studentischen Abgabe, Beispiel/- Musterlösungen, Lösungshinweise und Metadaten. Außerdem kann sie mehrere Tests beinhalten, die jeweils einen Typen und eine Test-Konfiguration besitzen. Eine task kann entweder nur als alleinige XML-Datei vorliegen oder als ZIP-Datei, die auch noch weitere Dateien beinhaltet. In diesem Fall muss auf der obersten Ebene aber die task.xml-Datei liegen, die die Aufgabe und somit auch den weiteren Aufbau der ZIP-Datei beschreibt.

2.2 Erweiterung des ProFormA-Austauschformats für variable Programmieraufgaben

Das ProFormA-Austauschformat ist zum gegenwärtigen Zeitpunkt nur dafür ausgelegt konkrete Aufgaben zu repräsentieren und bietet keine direkte Unterstützung für Variabilität in den Aufgaben. Allerdings gibt es diverse Erweiterungspunkte innerhalb des Formats, in denen eigene Werte und Strukturen eingefügt werden können, um eine einfache Anpassbarkeit an individuelle Anforderungen zu gewährleisten. In [3] wird ein Vorschlag für eine Notation zur Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben gemacht, der mit den Erweiterungsmöglichkeiten des ProFormaA-Formats kompatibel ist und dort auch auf diesem Wege eingesetzt wurde. Nachfolgend wird der Vorschlag zusammengefasst wiedergegeben.

Zunächst wird spezifiziert, auf welche Weise Variationspunkte innerhalb der task.xml-Datei kenntlich gemacht werden - dies geschieht mittels Platzhaltern, die in der Form *%vp{name_des_variationspunktes}* anstelle von konkreten Werten eingesetzt werden. So könnte zum Beispiel aus dem konkreten Satz *„Schreiben Sie ein Programm Counter.java“* der variable Satz *„Schreiben Sie ein Programm %vp{classname}.java“* werden. Sobald eine Wertbelegung für alle Variationspunkte vorliegt, kann mithilfe dieser Belegung aus der Aufgabenschablone eine Aufgabeninstanz generiert werden, indem die

[1] Verbundprojekt niedersächsischer Hochschulen und Vereine zur Verbesserung der Qualität der Lehre an diesen Hochschulen mittels digitaler Lern- und Lehrtechnologien. Siehe http://www.ecult-niedersachsen.de/

[2] Siehe http://www.ecult.me/was-bietet-ecult/proforma/

[3] https://github.com/ProFormA/taskxml/blob/master/whitepaper.md

vorgegebenen Platzhalter durch ihre entsprechende Wertbelegung ersetzt werden. Die so erstellte Aufgabeninstanz kann dann an Studierende zur Bearbeitung und auch an einen ProFormA-konformen Grader zur automatisierten Bewertung weitergereicht werden. Dies funktioniert, solange Variationspunkte nur in der task.xml-Datei existieren. Allerdings ist es auch durchaus möglich, dass weitere Artefakte innerhalb einer Aufgabe variiert werden sollen, wie z.B. Bibliotheken, zu verarbeitende Binärdateien oder allgemein Dateien, deren Format der Instanziierungssoftware nicht bekannt ist. Wie genau diese Artefakte instanziiert werden können, weiß letztlich nur der Grader selbst, da er sie später verarbeiten muss. Das genaue Zusammenspiel zwischen Instanziierungsdienst und Grader wird nicht genau spezifiziert, es werden aber einige kurze Vorschläge gemacht, wie die Kommunikation grundsätzlich ablaufen könnte.

Darauffolgend wird eine Notation in XML und Javascript vorgestellt, die in der Lage ist, alle möglichen Werte pro Variationspunkt und deren Kombinationen untereinander effizient darzustellen. Zunächst werden alle Variationspunkte und ihr Datentyp aufgelistet. Die Reihenfolge dieser Liste sagt aus, welcher im nachfolgenden beschriebene Wert zu welchem Variationspunkt korrespondiert. Nach dieser Aufzählung folgt eine Beschreibung der Wertemengen der einzelnen Variationspunkte. Hierfür gibt es acht bzw. neun verschiedene Arten von Knoten:

- Val: Repräsentiert einen einzelnen skalaren Wert
- Collect: Enthält mehrere skalare Werte; bei der Instanziierung wird nur einer der Werte ausgewählt
- Combine: Enthält mehrere skalare Werte; bei der Instanziierung werden diese zu einem Tupel zusammengefasst
- Range: Unter Angabe von erstem und letztem Element, sowie optional einer Schrittweite, kann hiermit sehr schnell eine (größere) Menge von Werten beschrieben werden; Bei der Instanziierung wird nur einer ausgewählt. Alternativ wäre ein Collect-Element mit vielen Werten möglich, Range ist lediglich eine Vereinfachung hierfür
- CollectGroup: Enthält eine Menge anderer Knoten; bei der Instanziierung wird nur einer der Knoten bzw. die darin enthaltene Wertemenge ausgewählt
- CombineGroup: Enthält eine Menge anderer Knoten; bei der Instanziierung werden diese Knoten bzw. die darin enthaltenen Wertemengen zu einem Tupel zusammengefasst
- Derivation: Enthält Javascript-Sourcecode, der eine Funktion „apply“ enthält, die als Eingabe ein Objekt aller bisher bestimmten Wertebelegungen erhält. Die Funktion kann daraus weitere Werte ableiten und entweder nur einen einzelnen Wert oder aber eine Menge von Werten zurückgeben

- Def/Ref: Mittels des Definitions-Knotens kann jedem Knoten eine ID zugeordnet werden. Im weiteren Verlauf kann der definierte Knoten mittels des Ref-Knotens referenziert werden. Zur Vermeidung von Redundanzen ist es somit möglich, sowohl skalare Werte (als Val-Knoten), aber auch ganze Teilbäume einmal zu definieren und anschließend an mehreren Stellen zu referenzieren

Die Bestimmung der Wertebelegungen erfolgt streng sequentiell von oben nach unten, da zwischen den einzelnen Variationspunkten häufig Abhängigkeiten bestehen. Der Einfachheit halber werden diese deswegen immer in der vorgegebenen Reihenfolge aufgelöst.

An jedem der Knoten kann außerdem eine Schlüsselumordnung stattfinden. Das bedeutet, dass die in den Kindknoten des umordnenden Knotens bestimmten Werte nicht in der zu Beginn vorgegebenen Reihenfolge den Variationspunkten zugeordnet werden, sondern die Zuordnung mit der in dem umordnenden Knoten angegebenen Schlüsselreihenfolge erfolgt.

Ähnlich wie bei der Instanziierung selbst, gibt es auch bei der Beschreibung bzw. Aufzählung der möglichen Werte Situationen, in denen nur der Grader selbst die Menge aller möglichen Werte bestimmen kann. Als Beispiel werden bei einer Java-Aufgabe die Menge aller installierten Zeichencodierungen oder bei einer SQL-Aufgabe die Menge aller Einträge einer Tabellenspalte genannt. Auch hier wird keine genaue Lösung vorgelegt, sondern lediglich einige kurze Vorschläge gemacht, wie dieses Problem grundsätzlich gelöst werden könnte.

2.3 Systemumfeld an der Hochschule Hannover

An der Hochschule Hannover werden bezüglich automatisiert-bewerteter Programmieraufgaben momentan der Grader Graja[2] für Java-Programmieraufgaben und der Grader aSQLg[5] für SQL-Aufgaben verwendet. Als Frontend dient in beiden Fällen das LMS Moodle[4], welches an der Hochschule auch unabhängig von den Programmieraufgaben sehr große Verwendung findet. Die Anbindung von Moodle an die beiden Grader erfolgt mithilfe der Middleware Grappa[4]. Eine auf die wesentlichen Komponenten heruntergebrochene Darstellung dieser Beziehungen findet sich in Abbildung 2.1. Grappa ist ein an der Hochschule Hannover im Rahmen des eCult-Projektes entwickelter Webservice, der zwischen einem oder mehreren LMS auf der einen Seite und einem oder mehreren Gradern auf der anderen Seite vermittelt, siehe Abbildung 2.2. Ohne eine solche Middleware, müsste für jede LMS-Grader-Kombination eine eigene Anbindung entwickelt werden; bei n Gradern und m LMSen ergäbe das $n * m$ zu entwickelnde Systeme. Mithilfe von Grappa kann diese Zahl auf $n + m$ verringert werden. In der Beispielabbildung müssen insgesamt nur fünf Anbindungen implementiert werden (da die Anbindung von Grappa an ein Grader-Plugin standardisiert ist). Ohne Grappa müssten die beiden LMSe an alle Grader einzeln angebunden werden, was sechs Anbindungen erfordern würde. Im

[4] https://moodle.de/

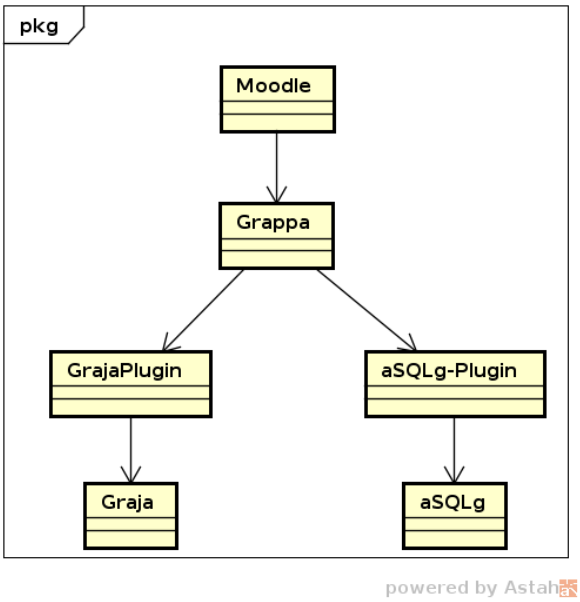

Abbildung 2.1: Vereinfachte Darstellung des Systemumfelds an der HS Hannover

kleinen Beispiel ist dies keine allzu große Ersparnis, allerdings kann dies auf potentiell alle existierenden LMSe und Grader angewendet werden, sodass sich in der Realität eine deutlich größere Ersparnis ergibt. Die Entwickler eines Graders müssen also nur eine einzige Anbindung an Grappa implementieren, sodass dem Grader dann alle LMSe zur Verfügung stehen, die ebenfalls an Grappa angebunden sind. Umgekehrt müssen die Entwickler eines LMS nur eine einzige Anbindung an Grappa entwickeln, damit dem LMS alle an Grappa angebundenen Grader zur Verfügung stehen. Die Anbindungen kümmern sich dann um die jeweiligen Eigenheiten des Graders/des LMS und Grappa selbst übernimmt allgemeine Aufgaben, die für eine Vermittlung immer erforderlich sind.

Der nachfolgende Workflow spiegelt zum Zeitpunkt des Entstehens dieser Arbeit nicht den tatsächlichen Workflow an der Hochschule wieder. Da Teile des ProFormA-Formats noch nicht komplett spezifiziert sind, verwendet Grappa noch eine eigens entwickelte Lösung. Es ist aber mindestens mittelfristig geplant Grappa so abzuändern, dass das ProFormA-Austauschformat unterstützt wird, weshalb in dieser Arbeit davon ausgegangen wird, dass diese Umstellung bereits stattgefunden hat.

Wenn eine Lehrperson in einem Kurs eine Aufgabe erstellen möchte, so kann er oder sie eine vom Moodle-Grappa-Plugin angebotene Programmieraufgabe anlegen. Beim Anlegen können typische Werte wie das Freischaltdatum ausgewählt werden, zusätzlich dazu gibt es noch ein Feld für den Upload einer task.zip-Datei. Wenn eine solche ProFormA-kompatible task.zip-Datei hochgeladen wird, extrahiert Moodle automatisch die daraus wichtigen Informationen und fügt sie der Aufgabe hinzu. Als wichtigstes Element ist hier sicherlich die Aufgabenbeschreibung zu nennen. Sind alle notwendigen Eingaben gemacht worden, kann die Aufgabe erstellt werden. Hierfür wird die task.zip-Datei an Grappa

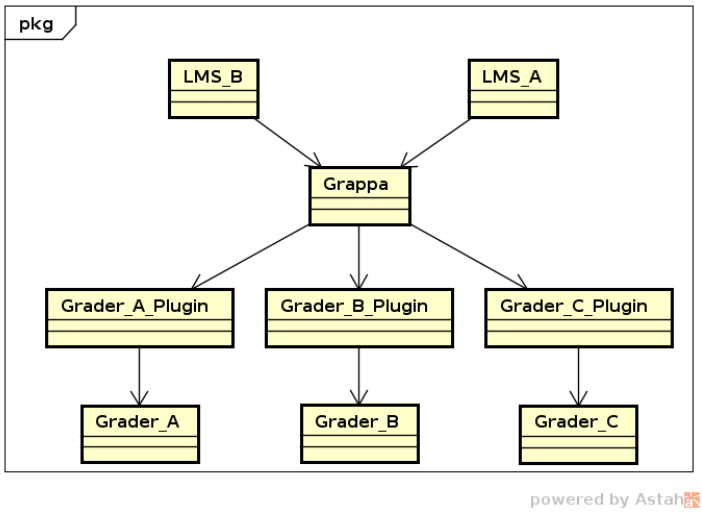

Abbildung 2.2: Vereinfachte Darstellung des Konzeptes von Grappa

übermittelt und dort persistiert. Ist bei diesem Vorgang kein Fehler aufgetreten, kann die Erstellung auch in Moodle abgeschlossen werden.

Möchten Studierende die Aufgabe bearbeiten, können sie sie im LMS aufrufen. Sobald sie eine Lösung erarbeitet haben und diese abgeben möchten, laden sie sie hoch und Moodle leitet sie, zusammen mit der Information, zu welcher Aufgabe die Abgabe gehört, an Grappa weiter. Grappa sucht die entsprechende task.zip-Datei heraus und leitet diese wiederum zusammen mit der studentischen Abgabe an den Grader weiter. Das Feedback zu der Aufgabe nimmt dann den umgekehrten Weg vom Grader über Grappa zurück zum Benutzer in Moodle.

2.4 Begriffe

Zum besseren Verständnis werden zunächst einige Begriffe definiert.
Eine *Aufgabenschablone* bezeichnet eine ProFormA-kompatible Aufgabe, die in irgendeiner Weise Variabilität besitzt. Sie kann nicht direkt an Studierende zur Bearbeitung gegeben werden. Hierfür muss zunächst eine *Aufgabeninstanz* aus der Aufgabenschablone generiert werden. Teilweise finden auch die englischen Begriffe *task template* und *task instance* Verwendung.
Ein *Variationspunkt* ist eine Art Parameter der Aufgabenschablone bzw. des Prozesses, der aus einer Schablone eine Instanz generiert. Variationspunkte haben einen Namen und einen Datentyp und können unterschiedliche Werte annehmen. Die konkreten Werte der Variationspunkte bestimmen, was genau für eine Instanz generiert wird.
Als *Wertebelegung* wird eine Menge bezeichnet, die jedem Variationspunkt einer Aufgabe einen konkreten Wert zuweist. In [3] wird dies als *composite variant (CV)* bezeichnet.

Ein *Artefakt* ist ein Element einer ProFormA-Aufgabe. Artefakte wären somit zum Beispiel die Description, ein File, ein Test oder eine ModelSolution.

3 Soll-Zustand

In diesem Kapitel wird der nach Abschluss der Arbeit angestrebte Zielzustand beschrieben und damit gleichzeitig herausgearbeitet, welche konkreten Ziele erreicht und was für Komponenten erstellt werden sollen. Zunächst folgt ein einführendes Beispiel, anschließend eine Beschreibung der Abläufe aus Nutzer-Sicht. Zuerst für den Dozenten und anschließend für Studierende. Im darauffolgenden Abschnitt wird geklärt, welche Verantwortlichkeiten die einzelnen Systeme haben, wie sie zusammenarbeiten und insbesondere wird diskutiert, an welchen Stellen was gecached werden könnte. Anschließend wird herausgearbeitet, in welcher Form Artefaktschablonen in Aufgabenschablonen vorkommen können. Zum Schluss werden die zu erstellenden Softwarekomponenten in den Systemkontext der Hochschule Hannover eingeordnet, geklärt, wie diese dort Verwendung finden sollen und welche der beschriebenen Komponenten Teil der Arbeit sind.

3.1 Variable Beispiel-Programmieraufgabe

Die nachfolgende variable Programmieraufgabe ist eine Beispiel-Aufgabe, auf die im weiteren Verlauf dieser Arbeit immer wieder verwiesen wird, um Konzepte, Ideen und Lösungsvorschläge zu verdeutlichen und die praktische Eignung der teils abstrakten Ausführungen zu belegen.

In der Aufgabe sollen mathematische Grundprinzipien nachprogrammiert werden. Entweder die Fibonacci-Zahlen oder die Berechnung einer Fakultät. Ein Variationspunkt entscheidet darüber, welche der beiden Aufgaben Studierende jeweils erhalten. Neben der entsprechend schablonisierten Aufgabenbeschreibung enthalten natürlich auch andere Elemente der task.xml-Datei Platzhalter, die während der Instanziierung ersetzt werden sollen. So gibt es z.B. statische Tests, die vom Grader beim Bewerten immer ausgeführt werden. Ein Beispiel wäre die Überprüfung, ob die geforderten Methoden in der abgegebenen Lösung vorhanden sind. Hierfür müssen die Platzhalter für die Methoden-Namen ersetzt werden, da die Methode in der einen Variante „berechneFakultaet„ und in der anderen „berechneFibonacci“ heißen soll.
Für die dynamischen Tests, die das abgegebene Programm auf seine korrekte Funktionsweise prüfen, ist in der Aufgabenschablone bzw. auch der späteren Instanz noch eine JAR-Datei enthalten, die alle für beide Aufgabenvarianten relevanten Tests enthält. Abhängig von den in der Aufgabeninstanz enthaltenen Test-Elementen für dynamische Tests, werden die jeweiligen Test-Methoden der JAR-Datei ausgeführt.

Unabhängig von der Wahl der Aufgabe, gibt es einen Variationspunkt „difficulty“, der die Schwierigkeit der Aufgabe zwischen „normal“ und „hard“ umschalten kann. Beim harten Schwierigkeitsgrad soll zusätzlich geprüft werden, ob eine korrekte Fehlerbehandlung bezüglich Falscheingaben besteht und negative Eingaben zurückgewiesen werden. Hierfür soll im Falle der harten Aufgabe im Rahmen der Instanziierung vom Grader noch ein weiteres Test-Element hinzugefügt werden, sodass der entsprechende Test der JAR-Datei ausgeführt wird.
Um die mathematischen Kenntnisse ggf. noch etwas aufzufrischen, enthält die Aufgabenschablone eine HTML-Datei inkl. Bildern, die die mathematisch zugrundeliegenden Konzepte noch einmal erläutern.

3.2 Abläufe aus Nutzer-Sicht

Grundsätzlich soll der Ablauf zum Erstellen einer variablen Programmieraufgabe ähnlich dem in Abschnitt 2.3 bereits beschriebenen Ablauf beim Erstellen einer nicht-variablen Programmieraufgabe sein. Somit wird auch eine vom LMS-Grappa-Plugin angebotene Programmieraufgabe angelegt und die allgemeinen Parameter eingestellt. Anstelle einer konkreten task-Datei wird aber eine Aufgabenschablone hochgeladen. Dies wird vom LMS automatisch erkannt und als „Meta-Datum“ in der Programmieraufgabe gespeichert. Sobald alle Einstellungen getroffen wurden, kann die Aufgabe erstellt werden.
Weiterhin ist geplant, dass es einen hochschulübergreifenden Aufgaben-Pool geben soll. In diesem Pool können alle Teilnehmenden Aufgaben - sowohl variable, als auch nicht-variable - ablegen und somit für andere freigeben und die freigegebenen Aufgaben anderer nutzen. Eine Anbindung an diesen Aufgaben-Pool kann direkt im LMS integriert werden, sodass dort eine Art „Aufgabenbrowser“ entsteht, mithilfe dessen der Pool durchsucht und eine Aufgabe direkt in das LMS übernommen werden kann.

Sobald die Aufgabe von einer Person zum ersten Mal aufgerufen wird, soll eine Aufgabeninstanz generiert werden. Dieser Ablauf unterscheidet sich jedoch zwischen Studierenden und Lehrpersonen. Für Studierende wird einfach eine zufällige Aufgabeninstanz erzeugt und ihnen zur Bearbeitung angezeigt. Diese Funktion ist für Lehrpersonen allerdings nicht praktikabel, da sie die Aufgabe vor Freischaltung häufig noch einmal testen wollen. Hierfür werden sie in der Regel eine konkrete Musterlösung haben, die auf genau eine Aufgabeninstanz passt. Aus diesem Grund wird Lehrpersonen beim ersten Aufrufen einer Aufgabe ein Dialog angezeigt, mit dem die Belegung der einzelnen Variationspunkte ausgewählt werden kann. Möglich wäre, dass ein oder mehrere Variationspunkte grader-spezifische Wertemengen besitzen. Das bedeutet, dass diese Wertemengen nur dem Grader bekannt sind. Bei diesen Variationspunkten wird zunächst ein Textfeld zur manuellen Eingabe angezeigt. Wird dieses ausgefüllt, überprüft der Instanziierungsservice beim Generieren automatisch, ob der eingegebene Wert legitim ist und erzeugt einen Fehler, falls dies nicht der Fall ist. Alternativ zur manuellen Eingabe kann pro Variationspunkt mit grader-spezifischer Wertemenge diese angefordert werden, so dass,

wie bei den anderen Variationspunkten auch, eine direkte Auswahl getroffen werden kann. Ist die Lehrperson mit der Wertebelegung zufrieden, wird mittels der ausgewählten Belegung eine Instanz generiert, die zur vorliegenden Musterlösung passt. Der soeben beschriebene Dialog kann von der Lehrperson jederzeit erneut aufgerufen werden, damit eine bestimmte Instanz zusammengestellt und generiert werden kann.

3.3 Interne Abläufe und Zusammenspiel der Komponenten

Der folgende Abschnitt beschreibt die Abläufe und Zusammenhänge zwischen den Systemen, die an (variablen) Programmieraufgaben beteiligt sind. Dies sind das LMS (ggf. mehrere), der Grader (ggf. mehrere), Grappa und der Instanziierungsservice. Da variable Programmieraufgaben zum gegenwärtigen Zeitpunkt noch nicht verwendet werden und es auch noch keine Planungen bezüglich der dafür notwendigen Änderungen an den beteiligten Systemen gibt, muss zunächst eine Anforderungsanalyse bezüglich einzelner Teilaspekte durchgeführt werden, die die Vor- und Nachteile der einzelnen Varianten diskutiert. Diese Diskussionen finden in den folgenden Abschnitten 3.3.1 bis 3.3.5 statt. Die Ergebnisse der einzelnen Diskussionen im Rahmen der Anforderungsanalyse werden als Fazit in Abschnitt 3.3.6 zusammengefasst.

3.3.1 Anforderungsanalyse des Problemfelds „(Zwischen)speichern der Aufgabenschablone“

Im Gegensatz zu den nicht-variablen Programmieraufgaben, soll die Aufgabenschablone nicht an Grappa übermittelt und dort gespeichert werden. Es empfiehlt sich aber, sie im LMS persistent abzulegen. Betrachtet werden sollten hierfür die beiden Fälle, wie eine variable Programmieraufgabe erstellt werden kann, bzw. wie die Aufgabenschablone in das LMS gelangt.
Im ersten Fall wird sie vom Lehrenden beim Erstellen der Aufgabe hochgeladen. In diesem Fall muss die Aufgabenschablone abgespeichert werden, da kein permanenter Zugang zur Festplatte der Lehrperson besteht, die Schablone aber jederzeit zur Verfügung stehen oder zumindest jederzeit abrufbar sein muss.
Der zweite Fall ist der, dass eine Aufgabenschablone aus dem Aufgaben-Pool in das LMS importiert wird. Unter der Annahme, dass die URL der Aufgabe sich nicht ohne weiteres ändert und der oder die Server des Aufgaben-Pools permanent erreichbar sind, wäre es denkbar, dass nicht die Aufgabenschablone selbst persistiert wird, sondern lediglich die URL, die auf die Aufgabe zeigt. Dies hätte den Vorteil, dass immer die aktuellste Version abgerufen wird, Änderungen somit direkt übernommen werden und es wird Speicherplatz eingespart. Fraglich ist aber, ob das Übernehmen der jeweils aktuellen Version in jedem Fall das gewünschte Verhalten ist. Denkbar wäre ein Szenario, in dem

recht große Änderungen an der Aufgabe gemacht werden, sodass alte Musterlösungen nicht mehr funktionieren oder die Lösung der Studierenden für die alte Version nicht kompatibel mit der neuen Version ist. In diesem Fall sollte die Lehrperson die Änderungen vorher sicherlich überprüfen und ein automatisches Aktualisieren sperren können. Dies würde die Anforderung an den Aufgaben-Pool stellen, dass auch alte Versionen einer Aufgabe vorgehalten werden. Weiterhin sind aktuelle variable Programmieraufgaben für den Grader Graja lediglich ca. 100 kB groß, selten auch bis zu 1 MB. Selbst wenn die Aufgaben also noch deutlich größer wären (10 MB, 100 MB), sind Speicherplatzbedenken an dieser Stelle bei heutigen Speicherkapazitäten nahezu auszuschließen. Weiterhin wird die Aufgabenschablone im LMS immer dann benötigt, wenn eine Aufgabeninstanz für Studierende generiert werden soll - dies passiert für alle Studierenden mindestens einmal, je nach Caching-Konzept aber durchaus auch öfters. Bei großen Aufgabenschablonen kann das mitunter zu einer recht starken Belastung für das Netzwerk führen. Auch würde das eine weitere Verzögerung bedeuten, die die Wartezeit der Studierenden zur Generierung einer Aufgabe weiter erhöht. Denkbar wäre auch eine Situation, in der die Aufgabe von der Lehrperson noch leicht angepasst werden soll, zum Beispiel bei der Bepunktung oder einzelnen Tests. Diese Änderungen lassen sich leichter und schneller an lokal abgespeicherten Daten durchführen, als die Änderungen separat zu speichern und bei jedem Abruf in die Aufgabenschablone zu integrieren. Es würde sich auch wieder die Frage stellen, ob die gemachten Änderungen überhaupt kompatibel mit einer neuen Version der Aufgabe sind, woraus sich abermals die Anforderung an den Aufgaben-Pool ableiten würde, dass dort mehrere Versionen einer Aufgabe gleichzeitig vorgehalten werden können.

3.3.2 Anforderungsanalyse des Problemfelds „Ansprechen des Instanziierungsservices“

Da der Instanziierungsservice grundsätzlich unabhängig von den ihn verwendenden Systemen ist, stellt sich die Frage, welche Komponente ihn ansprechen und verwenden soll. Dies könnte entweder das LMS, Grappa oder der Grader erledigen. Je weiter hinten in dieser „Kette“ das geschieht, desto mehr Systeme müssen mit Aufgabenschablonen umgehen können (Abbildung 2.1 zeigt die Abhängigkeiten der Systeme untereinander nochmal anhand des Systemumfelds an der HsH). Momentan gibt es nur Aufgabeninstanzen bzw. gar keine Schablonen. Falls das LMS sich um die Generierung der Instanzen kümmert, könnten Grappa und der Grader unverändert weiter benutzt werden, da sie weiterhin vom LMS nur Aufgabeninstanzen erhalten. Sollte Grappa die Generierung übernehmen, muss sowohl das LMS als auch Grappa mit Aufgabenschablonen umgehen können und für den Fall, dass es auf den Grader zurückfällt, muss auch dieser eine Schnittstelle dafür bieten.
Bezüglich des Anpassungsaufwandes wäre letzteres also das Worst-Case-Szenario, da in jedem Fall Grappa, zusätzlich aber auch noch alle LMS und Grader, die variable Programmieraufgaben unterstützen sollen, angepasst werden müssten. Im Besonderen ist

hier der Grader kritisch, da er damit explizit variable Programmieraufgaben unterstützen müsste - dies ist entsprechend dem Vorschlag in [3] aber gar nicht zwangsläufig nötig. Sondern nur dann, wenn der Grader noch spezielle grader-spezifische Instanziierungen durchführen möchte oder es grader-spezifische Wertemengen gibt.
Da Grappa als Middleware zwischen LMS und Grader gedacht ist, die insbesondere auch Funktionen übernehmen soll, die ansonsten redundant in mehreren Systemen implementiert werden müssten, wäre es durchaus denkbar, dass Grappa diejenige Komponente ist, die den Instanziierungsservice ansprechen soll. Allerdings müssten dann immer noch sowohl Grappa, als auch die LMSe angepasst werden. Abhängig davon, wie genau der Instanziierungsservice letztlich implementiert wird, ist es auch fraglich, ob das Ansprechen von Grappa für das LMS überhaupt leichter ist, als das Ansprechen des IS. Denn wenn der Instanziierungsservice eine sehr einfache Schnittstelle hat, die das LMS ebenso gut selbst ansprechen kann, wäre es eher kontraproduktiv, Grappa als Mittelsmann zu involvieren, da mit jedem zusätzlichen System in der Kette die Komplexität und Fehleranfälligkeit steigt. Ein weiteres Problem ist konzeptioneller Natur: Im Grunde soll Grappa nur zwischen mehreren LMSen und Gradern vermitteln, indem es Aufgaben(instanzen) und studentische Abgaben entgegen nimmt, diese an einen Grader weiterleitet und das entsprechende Feedback dazu zurück an das LMS gibt. Das Instanziieren von Aufgabenschablonen wäre eine weitere Aufgabe, für die es eigentlich nicht vorgesehen ist.
Somit bleibt noch die letzte, „einfachste“ (gemessen am Aufwand) Möglichkeit: Das LMS übernimmt die Koordinierung von Aufgabenschablonen, Aufgabeninstanzen und Instanziierungen. Dies hat den Vorteil, dass Grappa und Grader nicht angepasst werden und nicht einmal zwangsläufig etwas von Aufgabenschablonen wissen müssen. Weiterhin ist die Implementierung im LMS optional - falls keine variablen Programmieraufgaben benötigt werden, muss dies auch nicht implementiert werden.
Natürlich schließt dies nicht aus, dass die anderen beiden Varianten durchaus möglich sind und unter Umständen auch sinnvoll sein können. In dieser Arbeit wird allerdings vom zuletzt beschriebenen Szenario ausgegangen, in dem nur das LMS den Instanziierungsservice kennt und das in Abbildung 3.1 gezeigt ist. Die an den jeweiligen Schnittstellen ausgetauschten Daten werden in Abschnitt 3.3.6 noch genauer beschrieben.

3.3.3 Anforderungsanalyse des Problemfelds „Caching von Aufgabeninstanzen“

Ein weiteres Problem ist die Frage danach, an welcher Stelle Aufgabeninstanzen gecached werden sollen. Alle involvierten Systeme, das LMS, der Grader, Grappa und der Instanziierungsservice, arbeiten insofern selbstständig, als dass sie nur mit den Daten arbeiten, die sie an ihrer Eingangsschnittstelle erhalten. Es gibt keine zentrale Speicherverwaltung, keinen BUS oder ähnliches. Somit kann jedes der Systeme über ein eigenes, selbstständiges Caching-Konzept verfügen. Ein Gesamt-Konzept vorzuschreiben ist auch deswegen nicht sinnvoll, da die jeweiligen „Ausprägungen“ der Systeme, also konkrete LMS oder konkrete Grader, sich untereinander in Funktionsumfang und Funk-

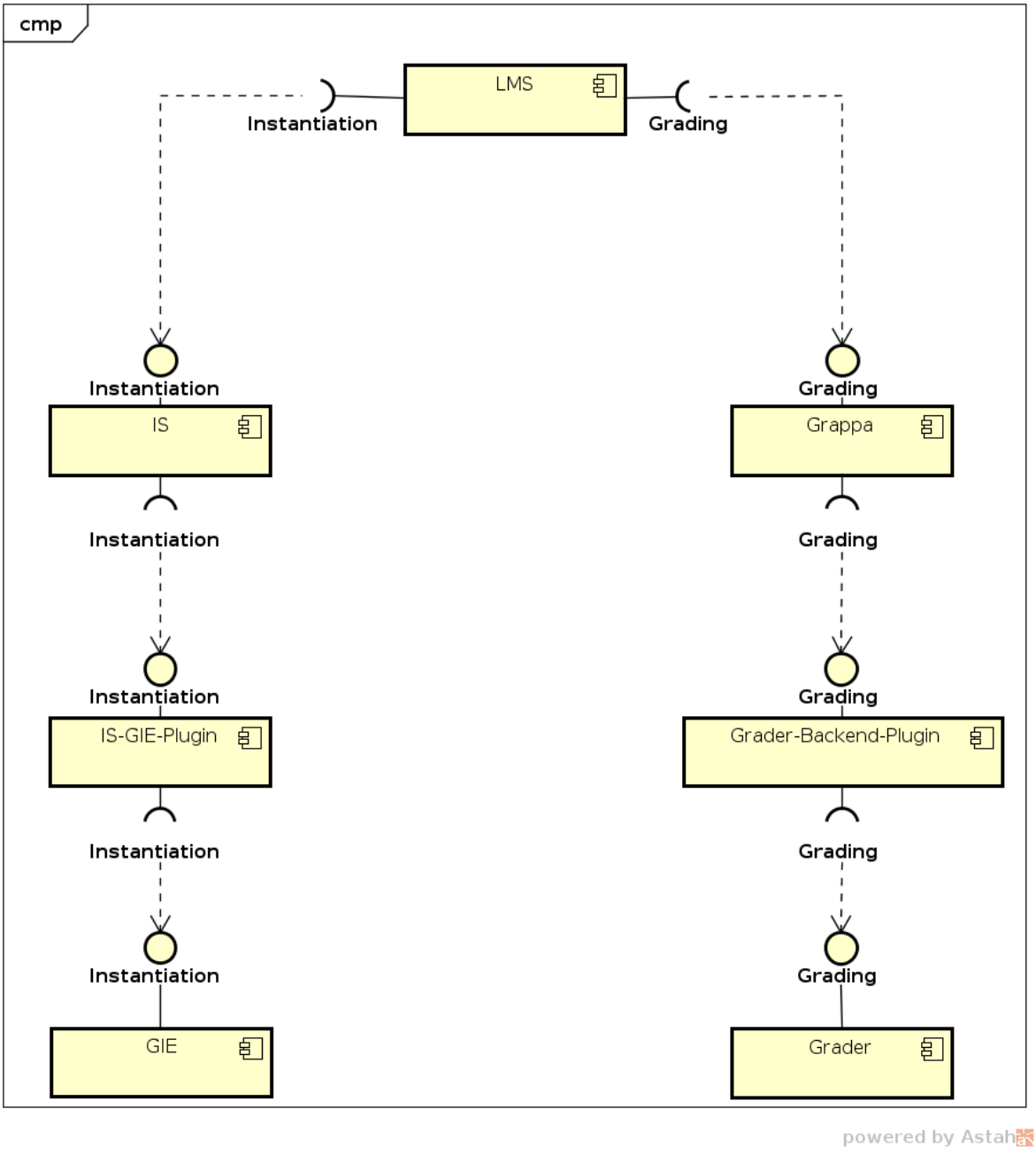

Abbildung 3.1: Abhängigkeiten der Komponenten untereinander (GIE = Grader instantiation engine, IS = Instanziierungsservice)

tionsweise sehr stark unterscheiden können. Aus diesen Gründen werden nachfolgend nur Vorteile und Nachteile des Cachings in den genannten Systemen diskutiert, aber keine Empfehlungen ausgesprochen, um eine Entscheidung diesbezüglich für eine konkrete Systemlandschaft zu erleichtern.
Zunächst wird das LMS betrachtet, da es in gewisser Weise Dreh- und Angelpunkt ist, da es sowohl mit dem Instanziierungsservice, als auch mit Grappa kommuniziert. In jedem Fall gespeichert werden, muss an dieser Stelle die Wertebelegung der Variationspunkte, um sicherzustellen, dass jederzeit wieder dieselbe Aufgabeninstanz generiert werden kann. Optional ist es, diese Aufgabeninstanz (zwischen) zu speichern. Dies hätte den Vorteil, dass Studierende nicht bei jedem Aufrufen der Aufgabe und jeder Abgabe ihrer Lösung warten müssten, bis eine Instanz generiert wurde. Lediglich beim ersten Aufrufen oder wenn der Cache ungültig ist, ließe sich eine Wartezeit nicht vermeiden. Sobald die Instanz das erste Mal generiert und im Cache abgelegt wurde, kann die Anfrage der Studierenden dann direkt aus diesem bedient werden. Neben der ständigen Verzögerung wäre ein weiteres Problem ohne Caching, dass immer wieder Serverressourcen belegt werden, um dieselbe Instanz zu generieren, die unter Umständen erst vor kurzer Zeit generiert wurde. Unter der Annahme, dass es zu einer Aufgabe hinreichend wenige Variationen gibt, bestünde mit dem Caching auch die Möglichkeit, Aufgabeninstanzen nur einmal abzuspeichern und später lediglich zu referenzieren. Sollten mehrere Studierende dieselbe Wertebelegung erhalten, müsste die Aufgabeninstanz dann nur einmal vorgehalten und könnte von allen zusammen verwendet werden. Ob diese Speicherplatzersparnis allerdings den Aufwand wert ist, sollte jeweils im Einzelfall diskutiert werden. Ein Nachteil beim Vorhalten der Instanzen ist sicherlich der Speicherbedarf. Anhand einer kurzen, fiktiven Beispielrechnung lässt sich aber festhalten, dass dies nicht allzu schwer wiegt. Angenommen in einem Kurs sind 100 Studierende. Es gibt über das Semester verteilt 10 Übungsblätter à 5 Aufgaben. Jede Aufgabe ist eine separate Programmieraufgabe, die im Schnitt 1 MB groß ist (also ca. das zehnfache der momentanen Durchschnittsgröße). Dies ergäbe pro Semester einen Speicherbedarf von $100 * 10 * 5 * 1MB = 5GB$ für Aufgabeninstanzen, falls alle Studierenden jede Aufgabe bearbeiten. Heutige Systeme sollten hiermit kein Problem haben.
Als nächstes wird der Instanziierungsservice betrachtet. Caching an dieser Stelle wäre sinnvoll, um potentiell redundante Arbeit zu vermeiden. Wird eine Instanz zu einer Aufgabenschablone mit einer Wertemenge angefragt, die genau so bereits vorher generiert und gecached wurde, könnte die Arbeit zum Generieren der Aufgabe entfallen und die Anfrage direkt aus dem Cache bedient werden. Gerade bei ressourcenintensiven Instanziierungen könnte dies einige Zeit und Rechenleistung sparen. Allerdings gibt es einige damit verbundene Schwierigkeiten. Zunächst ist es fraglich, wie oft eine Nachfrage für exakt dieselbe Instanz überhaupt stattfindet. Dies hängt maßgeblich von zwei Faktoren ab: Zum einen der Aufgabe selbst und wie viele Variationen es von ihr gibt und zum anderen, wie das Caching-Konzept im "vorgeschalteten"LMS gestaltet ist. Werden dort keine oder nur wenige Instanzen gecached, wird diese Nachfrage sicherlich häufig stattfinden. Werden alle oder viele Instanzen gecached, hängt es primär von der Anzahl der Aufgabenvariationen ab. Da das Vorhalten der Aufgabeninstanzen im LMS aber einige Vorteile hat und bereits wenige Variationspunkte mit kleinen Wertemengen zu

relativ vielen Variationen führen[1], ist tendenziell eher davon auszugehen, dass dieselben Aufgabeninstanzen eher selten mehrfach angefragt werden. Weiterhin stellt sich dann die Frage, wie lange Instanzen vorgehalten werden sollen. Grundsätzlich könnte dies in einer Art Konfigurationsdatei oder ähnlichem festgelegt werden. Allerdings gibt es Situationen, in denen die Inhalte des Caches durch äußere Umstände ungültig werden. Eine Lehrperson könnte zum Beispiel einen Fehler in der Aufgabenstellung bemerken und die Aufgabenschablone neu hochladen - in diesem Fall müsste eine Schnittstelle beim Instanziierungsservice vorhanden sein, die Managementfunktionen für den Cache anbietet.
Grappa selbst hätte in den meisten Fällen wohl nur noch die Aufgabe, Aufgabeninstanzen und studentische Abgaben entgegenzunehmen, diese an den Grader weiterzuleiten und anschließend das Feedback zurückzureichen. Die Erfahrung zeigt, dass die erste eingereichte Lösung in den seltensten Fällen absolut richtig ist. Die Wahrscheinlichkeit ist also hoch, dass dieselbe Aufgabeninstanz mehrfach vom LMS an Grappa und anschließend an den Grader übergeben wird. Das Zwischenspeichern von Instanzen in Grappa hat also den Vorteil, dass zumindest die Übertragung vom LMS an Grappa entfällt und das Netzwerk entlastet wird. Vorausgesetzt, LMS und Grappa sind sich einig darüber, wie eine Instanz im Cache referenziert werden kann und es besteht eine Fehlerbehandlung, falls die Instanz im Cache zum Beispiel nicht mehr gefunden werden kann. Dies würde beinahe dem Verhalten entsprechen, das Grappa momentan hat - mit dem kleinen Unterschied, dass momentan die im LMS hochgeladenen Aufgaben in Grappa gecached werden, dies in Zukunft aber die vom Instanziierungsservice generierten Aufgaben*instanzen* wären. Fraglich ist aber, wie groß die Entlastung des Netzwerks im Vergleich zu dem benötigten Aufwand ist, wenn Aufgabeninstanzen nur wenige 100 kB groß sind. Weiterhin sollte auch hier dann eine Managementschnittstelle für den Cache vorhanden sein, um diesen zum Beispiel ggf. leeren zu können. Außerdem würde dies in vielen Fällen eine doppelte Speicherbelastung bedeuten, da Instanzen sowohl im LMS, als auch in Grappa im Cache liegen. Wenngleich die Speicherbelastung in der Regel nicht allzu hoch sein wird, sei es doch trotzdem kurz erwähnt.
Der Grader benötigt die Aufgabeninstanzen, um anhand dieser die Bewertung der studentischen Abgaben vornehmen zu können. Der Vorteil vom Caching der Instanzen ist derselbe, wie im vorigen Abschnitt bei Grappa diskutiert. Mit dem Unterschied, dass zusätzlich zur Übertragung vom LMS zu Grappa, auch die Übertragung von Grappa zum Grader wegfällt, gilt somit auch dieselbe Argumentation, wie dort bereits erläutert.

3.3.4 Anforderungsanalyse des Problemfelds „Nachträgliches Ändern von Aufgaben“

Es wurde bereits angesprochen, dass das Ändern von Aufgaben ebenfalls Einfluss auf das Caching hat. Dies soll nun noch einmal kurz näher erläutert werden. Es gibt drei

[1] Bei vier voneinander unabhängigen Variationspunkten mit jeweils drei Werten, wären das bereits $3^4 = 81$ Variationen

„Eskalationsstufen“ in Bezug auf die vorgenommenen Änderungen.
Zunächst einmal wäre es möglich, dass die Änderungen rein „kosmetischer“ Natur sind, dass also zum Beispiel kleine Fehler in der Aufgabenbeschreibung behoben oder eine Formulierung präzisiert wurde. Im Beispiel könnte es sein, dass in der Aufgabenbeschreibung steht „Schreiben Sie eine Klsse, die [...]“, sodass hier nachträglich der Rechtschreibfehler in „Klasse“ behoben wird. In diesem Fall könnten die zwischengespeicherten Aufgabeninstanzen einfach verworfen werden, um anschließend mit der gespeicherten Wertebelegung eine aktualisierte Version der Instanz zu generieren. Dies könnte entweder direkt nach dem Verwerfen geschehen oder alternativ auch erst dann, wenn Studierende die Aufgabe das nächste Mal aufrufen. Notwendig wäre es in jedem Fall aber, die anderen Systeme darüber zu benachrichtigen, dass die entsprechenden Aufgabeninstanzen in ihrem Cache ungültig sind und verworfen werden sollen.
Eine tiefergehende Änderung wäre, dass zum Beispiel die Musterlösung oder die Dateien, mithilfe derer der Grader die Korrektheit der Abgaben prüft, einen Fehler enthalten und dieser korrigiert wurde. Im Beispiel könnte eine Test-Methode fälschlicherweise davon ausgehen, dass die Fibonacci-Folge 1-basiert implementiert wurde, obwohl sie laut Aufgabenstellung 0-basiert sein sollte. Auch dann müssen die Instanzen natürlich verworfen und neu generiert werden, das Problem ist allerdings, dass bereits abgegebene Lösungen potentiell nun anders bewertet würden. Falls dies automatisiert gelöst werden soll und die studentischen Abgaben gespeichert werden, kann nach dem Verwerfen einer Instanz direkt eine neue generiert und diese zusammen mit der Abgabe an Grappa zur erneuten Bewertung übergeben werden.
Die dritte Möglichkeit wäre, dass sich an der Aufgabe grundlegend etwas verändert hat, das auch die Variationspunkte und deren Wertemengen betrifft. Im Beispiel könnte es sein, dass die Lehrperson im Nachhinein der Meinung ist, dass die Berechnung einer Fakultät gegenüber der Fibonacci-Folge zu einfach ist und die Fakultäts-Variante durch die Stern-Brocot-Folge[2] austauscht, da diese eine größere Ähnlichkeit mit der Fibonacci-Folge hat. Sind lediglich weitere Werte hinzugekommen, stellt das kein Problem dar. Wurden aber Werte verändert, herausgenommen oder ganze Variationspunkte hinzugefügt, entfernt oder verändert, wie es im Beispiel der Fall ist, wäre das ein großes Problem. In diesem Fall muss nämlich nicht nur die Aufgabeninstanz verworfen werden, sondern ebenfalls die Wertebelegung, da sie inkompatibel mit der neuen Aufgabe ist. So würde die alte Wertebelegung zum Beispiel keinen Wert für einen neu hinzugefügten Variationspunkt beinhalten oder vielleicht einen Wert enthalten, der in der veränderten Wertemenge eines Variationspunktes gar nicht mehr enthalten und somit auch nicht mehr gültig ist, wie z.B. die Auswahl der Fakultäts-Variante. Wird dann eine neue Instanz - und somit auch eine neue Wertebelegung - generiert, ist diese potentiell eine ganz andere, als die vorige und die von Studierenden bereits erarbeitete Lösung wird sehr wahrscheinlich keine gültige Lösung der neuen Aufgabeninstanz sein. Automatisiert lässt sich dieses Problem somit im Allgemeinen nicht lösen. Denkbar wäre ein Algorithmus, der die alte Wertebelegung gegen die neue Variationspunktespezifikation abgleicht und so viele Werte wie möglich übernimmt, sodass eine mit der neuen Version kompatible Wertebelegung generiert wird,

[2]Wikipedia-Artikel der Stern-Brocot-Folge

die so nah an der alten liegt wie möglich. Selbst dann wäre die ursprüngliche Lösung aber im allgemeinen nicht immer kompatibel mit der neuen Aufgabeninstanz. Somit ist fraglich, ob der Aufwand für einen solchen Algorithmus gerechtfertigt ist oder es nicht einfacher wäre, dieses Problem manuell zu lösen, indem zum Beispiel für die neue Aufgabenversion eine ganz neue Programmieraufgabe im LMS angelegt wird und die Bearbeitung der alten Version freiwillig ist, Bonuspunkte bringt oder ähnliches.

3.3.5 Anforderungsanalyse des Problemfelds „Kommunikation vom LMS zum IS“

Auch sollte die Frage geklärt werden, welche Komponente im LMS den Instanziierungsservice anspricht. Die Antwort auf diese Frage hängt stark vom verwendeten LMS und möglicherweise bereits entwickelten Plugins ab, wird an dieser Stelle aber stellvertretend für das Moodle-Programmieraufgaben-Plugin[3] der Hochschule Hannover diskutiert. Zunächst scheint es sinnvoll, hierfür ein eigenständiges Plugin zu entwickeln, da eine extra Komponente angesprochen wird; die Verantwortlichkeiten wären somit klar verteilt und getrennt. Das Programmieraufgaben-Plugin kommuniziert mit Grappa, das IS-Plugin kommuniziert mit dem IS. Allerdings wird der Instanziierungsservice voraussichtlich nur vom Programmieraufgaben-Plugin benötigt und wäre ohne dieses somit ohne Nutzen. Umgekehrt benötigt das PA-Plugin den IS, um variable Programmieraufgaben anbieten zu können. Beide sind also stark voneinander abhängig. Ein weiterer Punkt, der für eine Zusammenlegung spricht, ist die Art und Weise, wie die Anbindung des Programmieraufgaben-Plugins an Grappa umgesetzt ist - diese erfolgt mittels einer Bibliothek, die mit Grappa kommuniziert und zwischen dem LMS und Grappa ggf. übersetzt. Dasselbe Konzept ließe sich auch auf den Instanziierungsservice übertragen, sodass hierfür ebenfalls eine entsprechende Bibliothek entwickelt werden könnte.

3.3.6 Fazit: Interne Abläufe im Soll-Zustand

Basierend auf der in den vorangegangenen Abschnitten erfolgten Anforderungsanalyse, wurde in Absprache mit den Betreuern der Abschlussarbeit der im Folgenden beschriebene Soll-Zustand festgelegt. Zum besseren Verständnis ist jeweils noch ein Sequenzdiagramm eingefügt, das den Ablauf darstellt. Diese entsprechen allerdings nicht genau der Implementierung, sondern dienen lediglich als Hilfestellung. Weiterhin zeigt Abbildung 3.1 noch einmal die Abhängigkeiten der Komponenten untereinander.
Wenn Studierende das erste Mal eine Aufgabe aufrufen, gibt das LMS, wie in Abbildung 3.2 dargestellt, dem Instanziierungsservice den Auftrag, zu einer gegebenen Schablone eine Aufgabeninstanz zu generieren. An welcher Stelle die hierfür notwendige Wertebelegung generiert werden kann, wird in Kapitel 5.1 diskutiert. Gegebenenfalls gibt der

[3] An anderer Stelle auch „Grappa-Plugin“ genannt, da es gegenwärtig nur mit Grappa kommuniziert. Grundsätzlich ist es aber von Grappa unabhängig.

IS die Aufgabenschablone zusammen mit der Wertebelegung zunächst noch einmal an den Grader, damit dieser grader-spezifische Werte ersetzen kann. Anschließend führt der Instanziierungsservice die Instanziierung im Rahmen seiner Möglichkeiten selbst durch und spricht ggf. noch das IS-Grader-Backend-Plugin an, um die restliche Instanziierung durchzuführen. Je nach Grader kann diese Funktionalität entweder direkt im Plugin liegen oder aber das Plugin ist nur eine Art „Adapter“, der den eigentlichen Grader-Service anspricht. Die fertige Aufgabeninstanz wird an das LMS zurückgegeben, abgespeichert und angezeigt.

In Abbildung 3.3 ist dargestellt, wie Studierende eine Lösung abgeben können. Die entsprechende Aufgabeninstanz wird abgerufen und zusammen mit der Abgabe an Grappa weitergereicht, das beides an den Grader zur Bewertung übergibt. Ist diese abgeschlossen, wird das Feedback wiederum zurück an das LMS gegeben und dort ebenfalls angezeigt.

Der Dialog, der einer Lehrperson beim ersten Aufrufen einer Aufgabe angezeigt wird, enthält möglicherweise grader-spezifische Wertemengen, welche angefordert und automatisch in den Dialog integriert werden können. Abbildung 3.4 verdeutlicht dies. Die Anforderung wird vom LMS an den Instanziierungsservice und von dort an das IS-Grader-Backend-Plugin weitergeleitet. Die Antwort mit der Wertemenge oder ggf. einem Fehler wird über denselben Weg zurück an das LMS gesendet.

3.4 Gestalt von Artefaktschablonen

Die einfache Art, Artefaktschablonen zu formulieren, wurde in Abschnitt 2.2 bereits näher erläutert. Anstatt konkreter Werte werden an einigen Stellen Platzhalter eingefügt, die dann vom Instanziierungsservice ersetzt werden. Diesen Artefakten sieht man in der Regel an, dass sie Schablonen sind. Für fortgeschrittene Szenarien und Instanziierungsmöglichkeiten kann es aber auch durchaus sein, dass zunächst nicht zwischen einer Artefaktschablone- und instanz unterschieden werden kann. Denkbar wäre zum Beispiel eine Komponente im Instanziierungsservice, die in irgendeiner Weise in der Aufgabenschablone enthaltene kleine Skripte ausführt, die ebenfalls Instanziierungen durchführen. Eine Möglichkeit wäre, dass ein Skript das fileref einer model-solution auf jeweils unterschiedliche Dateien „umbiegt“, abhängig von einem oder mehreren Variationspunkten. In diesem Fall könnte das fileref einen Standardwert haben und die model-solution ein vollständiges, gültiges model-solution-Element sein, dem man seinen Schablonen-Charakter nicht ansieht. Möglich wäre aber auch, dass das filerefs-Element in der model-solution fehlt (und die model-solution somit zunächst ungültig ist) und erst von dem Skript hinzugefügt wird. In Bezug auf die Beispielaufgabe könnte bezüglich des Test-Elements für schwierige Aufgaben mittels eines kleinen Skripts auch der umgekehrte Weg genommen werden: standardmäßig ist das Test-Element vorhanden und wird vom Skript immer dann entfernt, wenn der Variationspunkt „difficulty“ nicht den Wert „hard“ hat. Auch in diesem Fall würde man dem Test-Element nicht ansehen, dass es im Grunde eine Artefaktschablone ist und kein feststehendes Artefakt, das in jeder Instanz der Aufgabe vorhanden sein wird. Eine weitere Möglichkeit wäre, dass Attribute

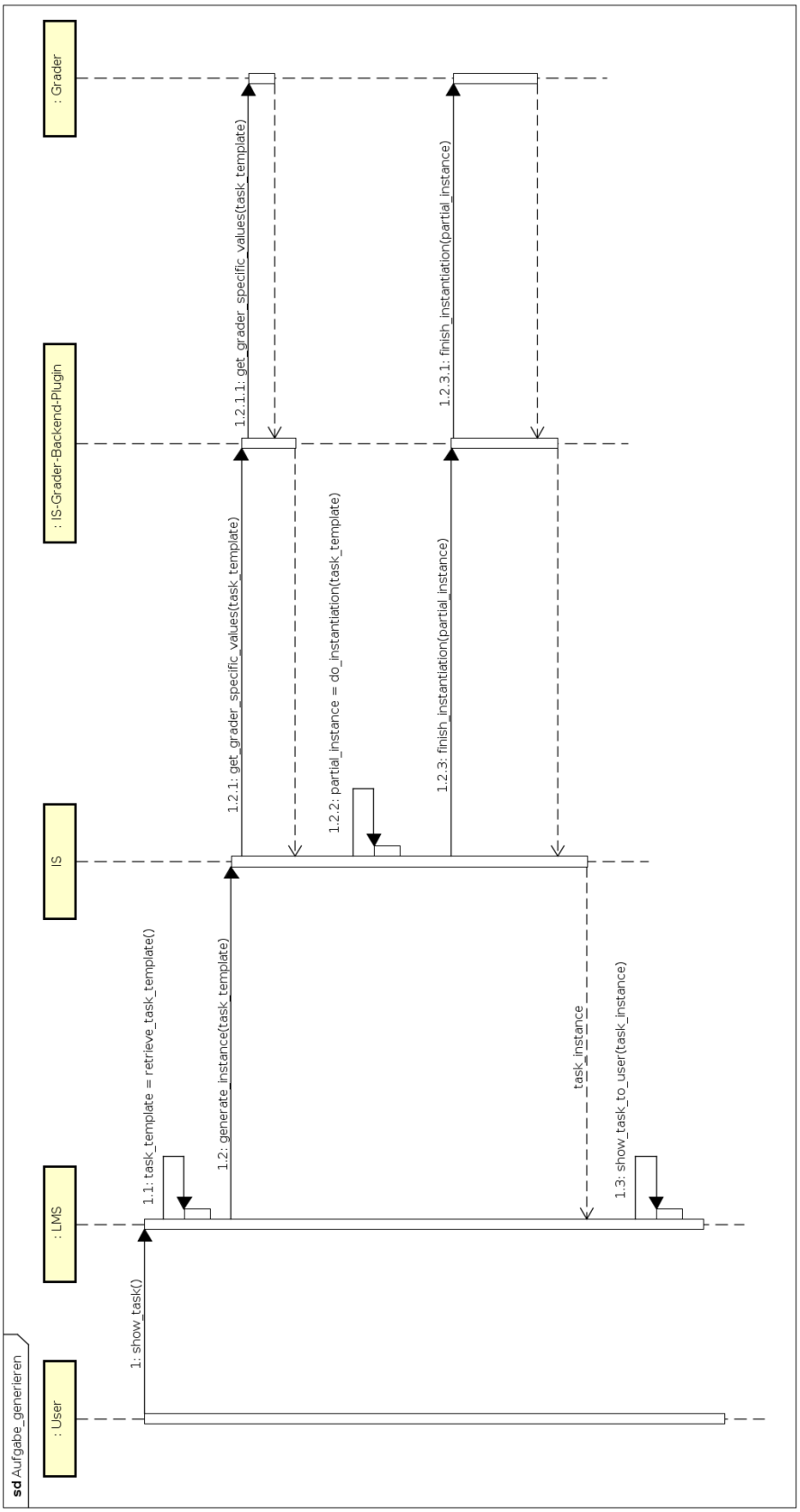

Abbildung 3.2: Erstmaliges Aufrufen einer Aufgabe durch Studierende

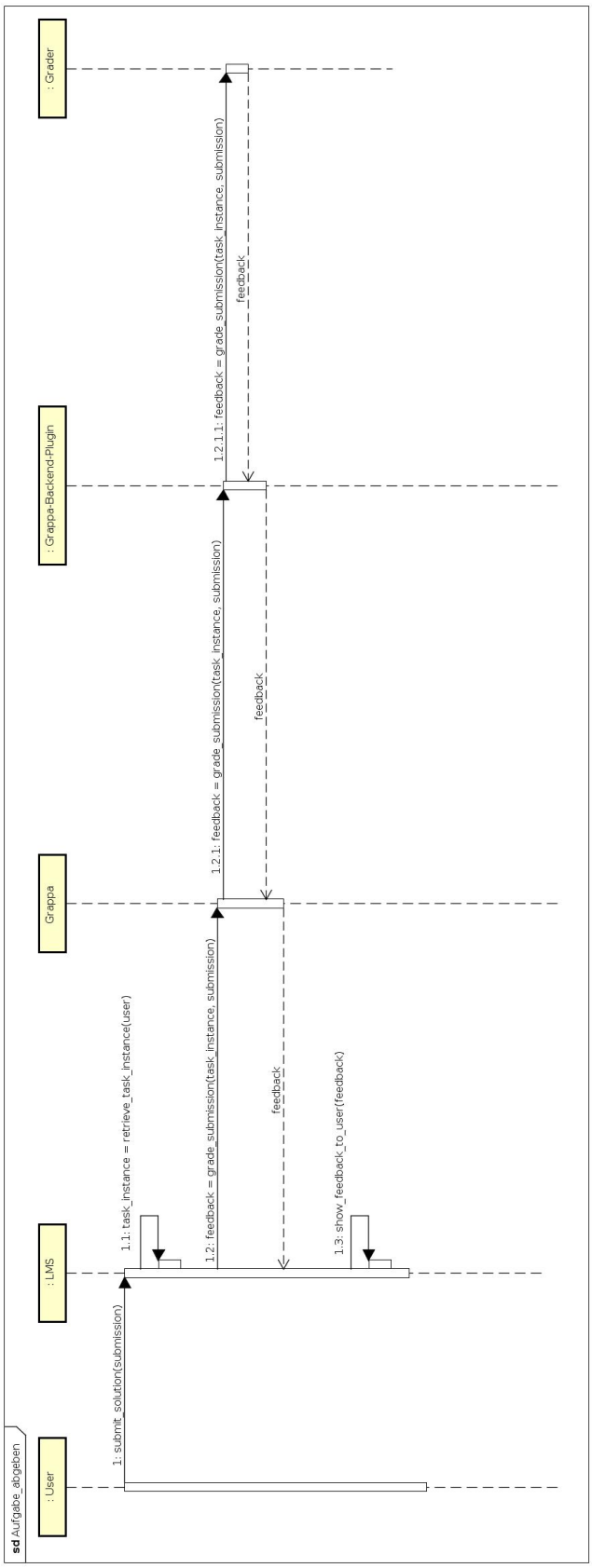

Abbildung 3.3: Abgeben einer studentischen Lösung

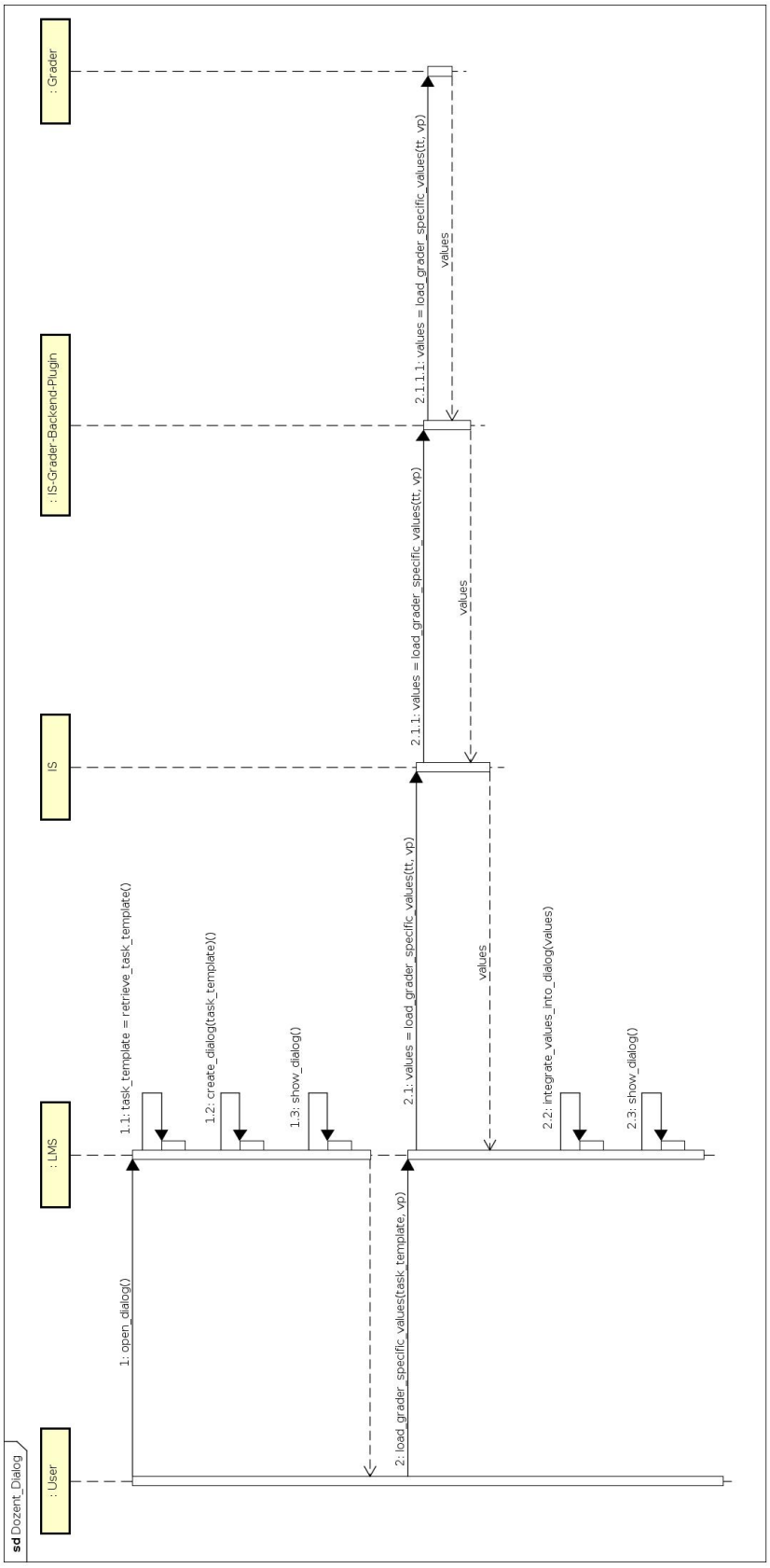

Abbildung 3.4: Anfordern von grader-spezifischen Wertemengen durch Lehrpersonen

oder Elemente selbst kleine Skripte enthalten, die einen Wert zurückgeben. Werden diese vom Modul ausgeführt, wird das Skript durch den zurückgegebenen Wert ersetzt. Der Instanziierungsservice sollte mit beiden Arten Artefaktschablonen anzugeben umgehen können (einfache und fortgeschrittene), um grundsätzlich für zukünftige Erweiterungen offen zu sein.

3.5 Integration an der Hochschule Hannover

An der Hochschule Hannover soll der Instanziierungsservice zunächst mit Grappa, dem LMS Moodle und dem Grader Graja zusammenarbeiten. Ein Moodle-Grappa-Plugin existiert bereits und kann um die erforderlichen Anforderungen erweitert werden. Die Erstellung eines mit dem IS kompatiblen Graja-Backend-Plugins wird ebenfalls Teil der Arbeit sein.

3.6 Ziele der Arbeit

Im Rahmen dieser Arbeit sollen folgende, im vorigen bereits teilweise beschriebene, Komponenten erstellt werden:

- Der Instanziierungsservice und eine Backend-Plugin-Schnittstelle
- Ein IS-Backend-Plugin für den Grader Graja
- Einfaches Konsolenprogramm, dem eine ProFormA-Aufgabenschablone übergeben wird und das dann den IS aufruft, um daraus eine Instanz zu generieren.

Hierfür soll der bereits vorliegende Grappa-Code soweit wie möglich wiederverwendet werden, sodass im Idealfall eine Bibliothek entsteht, die sowohl vom IS, als auch von Grappa genutzt werden kann.

Nicht Teil der Arbeit ist die Anpassung des Moodle-Grappa-Plugins und von Grappa selbst.

Das Abrufen von grader-spezifischen Wertemengen soll zwar entworfen, aber nur optional mit implementiert werden.

Zunächst sind für den Instanziierungsservice nur einfache Aufgaben, wie das Suchen und Ersetzen von Platzhaltern geplant. Vorschläge wie das in Abschnitt 3.4 angesprochene Skripte-Modul sind zunächst nur Ideen, die nicht umgesetzt, für einen möglichen Entwurf aber im Hinterkopf behalten werden sollen, um dahingehende Erweiterungen einfacher zu ermöglichen.

4 Methodik und Arbeitsprogramm

Da die in dieser Arbeit zu erstellenden Softwarekomponenten für den realen Praxiseinsatz vorgesehen sind, war ein großer Teil der Anforderungen bereits zu Beginn festgelegt. Die weitere Abstimmung und Klärung von Detailfragen bezüglich der Anforderungen erfolgte in Abstimmung mit Prof. Dr. Robert Garmann, da dieser an der Einführung von variablen Programmieraufgaben arbeitet. Anschließend wurde geklärt, in welchem Systemkontext die zu erstellende Software zum Einsatz kommen wird, um die Verantwortlichkeiten zu verteilen und zu klären, ob sich hieraus möglicherweise noch zusätzliche Anforderungen ergeben, siehe Kapitel 3.

Nachdem die Anforderungen endgültig geklärt waren, wurden mit Robert Garmann und Peter Fricke zunächst unterschiedliche Lösungsansätze diskutiert. Die Ergebnisse dieser Diskussionen wurden anschließend in Modelle (UML-Diagramme und XML-Schemata) überführt und noch ein weiteres Mal besprochen, vgl. Kapitel 5.

Daraufhin wurde mit der Umsetzung des besprochenen Entwurfs und der Implementierung begonnen, Details hierzu in Kapitel 6. Währenddessen wurden, soweit sinnvoll und notwendig, Tests durchgeführt, um die korrekte Funktionsweise aller Komponenten schon während der Entwicklung sicherzustellen. Weiterhin wurde nebenher ein simples Test-Frontend entwickelt, mithilfe dessen der Instanziierungsservice manuell getestet werden kann, indem ihm eine Aufgabenschablone übergeben wird und eine Aufgabeninstanz zurückgeliefert werden soll. Nach Fertigstellung der Implementierung wurden noch diverse Systemtests durchgeführt, dokumentiert in Kapitel 7, um eine korrekte Funktionsweise zu gewährleisten.

5 Lösungsmöglichkeiten

Im folgenden Kapitel werden unterschiedliche Möglichkeiten diskutiert, wie einzelne Komponenten des Instanziierungsservices bzw. Abläufe während der Instanziierung realisiert werden können.

5.1 Generieren einer Wertebelegung

Da der Instanziierungsprozess abhängig von einer konkreten Wertebelegung ist, wird zunächst betrachtet, wie diese erstellt werden könnte. Zum einen wäre es möglich, sie als weiteren Eingangsparameter zu definieren, so dass immer die Aufgabenschablone zusammen mit einer Wertbelegung übergeben wird, zum anderen wäre es auch möglich, dass der Instanziierungsservice die Bestimmung der Wertbelegung selbst vornimmt. Dies hätte den Vorteil, dass die Funktionalität zur Erstellung nur einmal implementiert werden müsste und Systeme, die den IS nutzen, sich nicht darum kümmern müssten. Da die Spezifikation der Variabilität allgemein spezifiziert ist, wäre das auch durchaus möglich. Allerdings könnten auf diese Weise nur rein zufällige Wertbelegungen generiert werden, da der IS über keinerlei zusätzliches Wissen verfügt; er kennt lediglich die Aufgabenschablone und generiert dazu eine zufällige Instanz. Hier wäre die erste Variante im Vorteil: Wenn das LMS für die Generierung zuständig ist, können noch weitere Parameter mit einfließen. So kennt das LMS zum Beispiel den aktuellen Leistungsstand der Studierenden (oder zumindest eine Annäherung dessen in Form der bisher erbrachten Leistungen) und könnte darauf basierend den Schwierigkeitsgrad der Aufgabe abstimmen. Eine entsprechende Schnittstelle beim IS, die Aufgabenschablone und Wertebelegung entgegen nimmt, muss ohnehin existieren, da Lehrpersonen die Möglichkeit haben sollen, sich eine individuelle Instanz zu erstellen, vgl. Abschnitt 3.2. Um den dort beschriebenen Dialog zu realisieren, muss im LMS auch das entsprechende Wissen vorhanden sein, um die Spezifikation der Variabilität verarbeiten und im Auswahldialog darstellen zu können. Falls nicht ohnehin bereits vorhanden, könnte darauf aufbauend die Funktionalität zur automatischen Generierung einer Wertbelegung erstellt werden. Wenn eine solche Komponente, zur Verarbeitung und automatischen Generierung, in einer Sprache wie Javascript realisiert wird, die von sehr vielen Systemen, gerade im Web-Umfeld, in dem der Großteil der LMS angesiedelt sind, verstanden und interpretiert werden kann, müsste diese Komponente auch auf diese Weise nur einmal realisiert und könnte in vielen Systemen wiederverwendet werden.

5.2 Verantwortlichkeiten von Instanziierungsservice und Grader

Grundsätzlich könnte der Grader die Instanziierung vollständig übernehmen und der Instanziierungsservice lediglich als Mittelsmann für unterschiedliche Grader agieren. Es stellt sich allerdings die Frage, ob ein solches Konzept sinnvoll wäre und, falls nicht, welche Verantwortlichkeiten dem IS selbst zufallen könnten. Das wären Aufgaben, die für jede einzelne Aufgabenschablone gleich sind und somit von einer einzelnen Komponente erledigt werden können. Da alle Aufgabenschablonen die Variationspunkte und Wertebelegungen gemein haben, könnte der IS die Platzhalter für die Variationspunkte mit der jeweiligen Wertbelegung austauschen. Abhängig davon, wie die Aufgabenschablone konzipiert ist, könnte dadurch bereits ein großer Teil der Instanziierung abgedeckt werden. Weitere Verantwortlichkeiten an den IS zu übergeben gestaltet sich schwierig, da hierfür eine allgemeine Grundlage fehlt, wie sie mit den Wertebelegungen gegeben ist. Aufgabenspezifische Tests hinzufügen, Binärdateien in proprietären Formaten instanziieren oder dynamisch Musterlösungen generieren kann zum Beispiel nur der Grader, da er, zumindest potentiell, mehr Wissen über die Aufgabe hat, als der IS. So könnten alle Aufgaben für diesen Grader zum Beispiel einer bestimmten Struktur folgen, anhand derer er entscheiden kann, was zu tun ist. Hierfür wäre es natürlich notwendig, die gesamte Aufgabenschablone an den Grader zu übergeben.
Ein weiterer Punkt, der vom Grader übernommen wird, ist das Zurückliefern von graderspezifischen Wertemengen. In diesem Fall ist der IS tatsächlich lediglich der Mittelsmann, der Anfragen hierzu entgegen nimmt und diese unbearbeitet an das Grader-Backend-Plugin weiterleitet. Der IS selbst kann die Anfrage nicht bearbeiten, da ihm hierfür das Wissen fehlt. Je nach Grader wäre es dann entweder möglich, die Anfrage direkt im Backend-Plugin zu verarbeiten und zu beantworten oder sie wird von diesem nur an den Grader weitergereicht, sodass dieser selbst antworten kann.

5.3 Instanziierung durch den Instanziierungsservice

Der Instanziierungsservice kann die Instanziierung der Dateien vornehmen, die er als Text-Dateien interpretieren kann, um hier die notwendigen Suchen-Und-Ersetzen-Operationen durchzuführen. Es stellt sich die Frage, woher er weiß, welche Dateien in der übergebenen zip-Datei das sind. Eine ist sicherlich die task.xml-Datei selbst, da die Wahrscheinlichkeit sehr hoch ist, dass dort Variationspunkte vorhanden sind. Insbesondere, da dort die Aufgabenbeschreibung enthalten ist und diese in den meisten Fällen wohl von Instanz zu Instanz leicht unterschiedlich ausfallen wird. Die task.xml-Datei ist aber in dieser Hinsicht gewissermaßen ein Sonderfall und wird in Abschnitt 5.5 genauer behandelt.
Bei den restlichen Dateien könnte der IS explizit oder implizit vorgehen. Zum einen könnte er implizit annehmen, dass alle Dateien instanziiert werden sollen und somit versuchen, diese Operation auf alle anzuwenden. Allerdings führt das bereits beim Einlesen

zu Problemen: Es werden auch nicht-Text-Dateien eingelesen, bei denen es dann zu einem Fehler kommen kann, da zum Beispiel die Lösungs-JAR-Datei in der Beispiel-Aufgabe nicht sinnvoll als Text-Datei interpretiert werden kann. Selbst wenn es sich bei der gerade zu verarbeitenden Datei tatsächlich um eine Text-Datei handelt, weiß der IS nicht, wie diese codiert ist. Im ProFormA-Standard ist lediglich festgelegt, dass die task.xml in UTF-8 codiert sein soll, ansonsten gibt es aber keine Vorgaben. Es wäre zwar möglich, den IS die Codierung „raten“ zu lassen, indem er einfach versucht, die Datei mit unterschiedlichen Codierungen zu lesen, allerdings ist auch das fehleranfällig.
Selbst wenn es funktioniert, gibt es ein weiteres, fachliches Problem. Denn es ist nicht gesagt, dass tatsächlich alle Dateien instanziiert werden sollen. Es wäre gut möglich, dass der IS dies bei einigen zwar tun könnte, aber nicht tun soll, da der Grader die nicht-instanziierte Datei für seine Instanziierung benötigt. Wenn er die Platzhalter zum Beispiel als Anhaltspunkt für seine eigene Instanziierung braucht und sie dann anschließend selbst ersetzt. Weiterhin wäre es denkbar, dass hierbei versehentlich etwas „instanziiert“ wird, das nicht zur Instanziierung vorgesehen war, sondern vom IS fälschlicherweise so interpretiert wurde. Wenn ein Platzhalter für einen Variationspunkt „img“ zum Beispiel „<img“ heißt und der IS in der Beispielaufgabe versucht, die HTML-Datei zu instanziieren, würde er die dort enthaltenen HTML-Image-Tags ersetzen und somit durch das „instanziieren“ erheblichen Schaden anrichten.
Alternativ zum impliziten Vorgehen wäre das explizite Vorgehen. Hier könnte man wiederum entweder explizit angeben, welche Dateien instanziiert werden sollen oder aber, welche Dateien nicht instanziiert werden sollen. Ein explizites Exkludieren hätte zwar den Vorteil, dass versehentliche bzw. fälschliche Ersetzungen, wie die in der HTML-Datei, nicht mehr vorkommen, da der IS angewiesen werden könnte, diese zu ignorieren, allerdings bestünde weiterhin das Problem, dass die Codierungen nicht bekannt sind und es könnte weiterhin passieren, dass versehentliche Ersetzungen in den richtigen Dateien vorgenommen werden. Also zum Beispiel eine Datei zwar instanziiert werden soll, aber deren Format auch gleichzeitig ein Schlüsselwort enthält, das einem Platzhalter entspricht und dieses Schlüsselwort somit nicht ausgetauscht werden sollte. So könnte es zum Beispiel sein, dass in der HTML-Datei noch einmal die Aufgabenbeschreibung wiederholt werden soll, sodass auch diese Datei instanziiert werden müsste. Da das allgemeine Platzhalter-Format aber aus irgendwelchen externen Gründen möglicherweise nicht verändert werden kann, sondern „<*Platzhaltername*“ bleiben muss, einige Platzhalter aber mit HTML-Tags kollidieren würden, müsste in diesem Fall darauf verzichtet werden, die Aufgabenbeschreibung dort mit aufzunehmen.
Das explizite Inkludieren hingegen hätte diese Probleme nicht. Es könnte jede zu instanziierende Datei einzeln angegeben werden und zusätzlich dazu die Codierung. Um das Problem mit fehlinterpretierten Platzhaltern zu lösen, könnte zusätzlich pro Datei noch das Platzhalter-Format angegeben werden. Dies ermöglicht, die Platzhalter pro Datei festzulegen und sie möglichen Schlüsselwörtern oder ähnlichem anzupassen. Somit könnte dann auch nur für die HTML-Datei ein abweichendes Platzhalter-Format festgelegt werden, sodass die Aufgabenbeschreibung dort mit aufgenommen werden kann. Um Schreibarbeit zu sparen, wäre es weiterhin denkbar, ein allgemeingültiges Platzhalter-Format anzugeben, das pro Datei überschrieben werden kann. Ohne Angabe

eines Formats bei der Datei wird das Allgemeingültige genommen, ansonsten das Überschreibende.
Ein weiterer Punkt, der zu klären ist, ist der, ob für das eigentliche Instanziieren, das „Suchen/Ersetzen“, eine bereits existierende Template-Engine verwendet, oder diese Funktionalität selbst implementiert wird. Eine Template-Engine hätte den Vorteil, dass bereits existierende Funktionen direkt verwendet werden könnten, so zum Beispiel die Fehlerbehandlung, Beachtung von Rand-Fällen oder Zusatzinformationen des Ersetzungs-Prozesses. Allerdings würde man sich von einer weiteren Komponente abhängig machen, in die zunächst auch eine Einarbeitung erfolgen muss. Des weiteren sollte untersucht werden, ob die bereitgestellten Funktionalitäten überhaupt benötigt werden.

5.4 Instanziierung durch den Grader

Je nach eingesetztem Grader unterscheiden sich die von ihm zur Instanziierung bereitgestellten Funktionalitäten. Es besteht die Möglichkeit, dass er immer genau weiß, was er bei jeder Aufgabenschablone zu tun hat oder es zumindest selbst ableiten kann, zum Beispiel anhand der Struktur der Aufgabe. Dieses Wissen kann man allerdings nicht allgemeingültig für alle Grader voraussetzen. Denkbar wäre ein Grader, der zwar Instanziierungsfunktionalitäten bietet, hierfür allerdings eine Art Hilfestellung benötigt, die angibt, was genau instanziiert werden soll. Für solche schon existierenden Grader ist diese Hilfestellung vermutlich bereits in einer grader-spezifischen Form in der Aufgabenschablone vorhanden. Ein Grader für die Beispielaufgabe könnte zum Beispiel in den grading-hints nach einem Element *addTestDependingOnVpValue* mit den Attributen *vpName* und *vpValue* suchen, dass das entsprechende Test-Element enthält und ihm sagt, dass er das Element den Tests hinzufügen soll, wenn der angegebene Variationspunkt den angegebenen Wert hat. Im Rahmen der Entwicklung eines standardisierten Instanziierungsservices, wäre es somit sinnvoll, auch ein Datenformat für eine solche Hilfestellung zu spezifizieren.
Denkbar wäre dies zum Beispiel als Liste von Informationsblöcken, wobei jeder Informationsblock ein vom Grader zu instanziierendes Artefakt betrifft. Die Elemente in der Liste müssen mindestens solche Attribute enthalten, die für eine eindeutige Identifizierung des referenzierten Artefakts benötigt werden. Darüber hinaus wären noch weitere Attribute denkbar, die allgemein für jede Art von referenziertem Artefakt sinnvoll sind, wie zum Beispiel die optionale Angabe eines Platzhalter-Formats, wie sie in Abschnitt 5.3 schon beschrieben wurde. Ein Problem ist, dass die grader-spezifischen Listen vermutlich grader-spezifische Elemente oder Attribute für die Artefakte enthalten, wie es auch im Beispiel der Fall ist. Diese können nicht alle explizit in eine Spezifikation aufgenommen werden, da es schlicht zu viele sind, um alle denkbaren Grader zu unterstützen. Die Codierung einer Datei zum Beispiel ist noch recht naheliegend, allerdings könnte einen Grader möglicherweise auch das letzte Änderungsdatum einer Datei, die Version eines Tests oder vorab bereits die Größe einer externen Ressource interessieren. Ebenso könnte er eine beliebige Menge von eigenen Elementen definiert haben, die ihm genaue Anweisungen

geben, wann, was, wie instanziiert werden soll. Da eine Spezifikation aller möglichen Optionen somit vermutlich nicht möglich ist, könnte alternativ spezifiziert werden, dass neben den erforderlichen Attributen und Elementen auch beliebige andere hinzugefügt werden können. Die Liste würde dann als gültig anerkannt werden, wenn alle erforderlichen Elemente alle erforderlichen Attribute enthalten. Optionale weitere Attribute und Elemente werden nicht direkt verarbeitet, sondern einfach weitergereicht. Auf diese Weise wird ein Grundgerüst für alle Grader bereitgestellt, das bei Bedarf aber noch beliebig erweitert und ggf. an Grader angepasst werden kann. Um zum einen unnötige Arbeit zu ersparen und zum anderen semantischen Missverständnissen vorzubeugen, könnte die ganze Liste auch als optional spezifiziert werden. Das Weglassen würde dann nicht als „der Grader führt keine Instanziierung durch“ interpretiert werden, sondern als „der Grader weiß selbst, was er instanziieren muss“.
Grundsätzlich wäre es erstrebenswert, das eben beschriebene Datenformat für den Grader, ebenso wie die in Abschnitt 5.3 beschriebene Liste für den Instanziierungsservice, in das allgemeingültige ProFormA-Format zu übernehmen, um sie nicht nur als Erweiterung, sondern richtigen Teil des Formats angeben zu können. Der Vorteil daran wäre, dass auch andere ProFormA-kompatible Werkzeuge die Aufzählungen direkt unterstützen würden, um sie zum Beispiel in einem Dialog noch besser editierbar zu machen.

Es stellt sich die Frage, was genau der Grader mit den instanziierten Artefakten macht bzw. was genau der Grader an den Instanziierungsservice zurückgibt. Zum einen wäre es denkbar, dass der Grader selbst die Artefakte in die vom Instanziierungsservice bereits teilinstanziierte Aufgabenschablone integriert und somit eine ganze Aufgabeninstanz zurückliefert. Zum anderen wäre es möglich, dass der Grader lediglich die instanziierten Artefakte an den IS zurückgibt und dieser sie selbst integriert und somit die Instanziierung abschließt.
Grundsätzlich wäre die erste Option zwar möglich, allerdings ist die Integration eine allgemeine Operation, die für alle Aufgabenschablonen gleich ist. Es wäre somit eine Funktionalität, die in allen Gradern redundant implementiert werden müsste, weshalb die zweite Variante in diesem Fall besser geeignet ist. Somit muss eine Liste von instanziierten Artefakten übertragen werden, deren genaue Spezifikation zur Diskussion steht. Eine Möglichkeit wäre, die Elemente in der Liste so allgemein wie möglich zu gestalten, um sie einfach erweitern zu können. Alle Elemente könnten zum Beispiel eine ID, einen Typ und einen Inhalt haben. Der Typ gibt an, um was für ein Artefakt es sich handelt, eine Datei zum Beispiel oder einen Test und im Inhalt steht dann der entsprechende Test oder die Datei im ProFormA-XML-Format. Werden später weitere Möglichkeiten für Artefakte hinzugefügt, kann einfach ein neuer Typ hinzugefügt werden. Allerdings ist dieser Ansatz nicht typsicher, es kann nicht sichergestellt werden, dass der Inhalt eines übergebenen Elements vom Typ „Test“ tatsächlich einen (gültigen) Test enthält. Sehr wahrscheinlich wird der IS den Inhalt irgendwann mittels unmarshalling in eine Java-Objektstruktur überführen, um die Integration der Artefakte auf Java-Objektebene durchzuführen. Spätestens an dieser Stelle muss das Element vom Typ „Test“ dann einen gültigen Test beinhalten, da es sonst zu einem Fehler kommt. Weiterhin ist das Argument der Erweiterbarkeit nur bedingt zu gewichten, da es eher unwahrscheinlich

ist, dass weitere Artefakttypen hinzukommen werden. Denkbar wäre es zwar durchaus, allerdings wird das nicht der Regelfall sein, sodass eine „größere“ Anpassung dann durchaus vertretbar wäre.
Ein anderer Ansatz wäre somit, die Typsicherheit direkt an der Schnittstelle sicherzustellen und in der Rückgabeliste nur bestimmte Typen zu erlauben. Für Artefakte innerhalb der task.xml-Datei könnten sogar direkt die entsprechenden Definitionen der Typen aus dieser übernommen werden. Somit ist sichergestellt, dass nur gültige Artefakte zurückgegeben werden. Auf diese Weise wäre es auch einfach möglich, sowohl bereits bestehende Artefakte zu ersetzen, als auch neue Artefakte hinzuzufügen. Der Instanziierungsservice kann prüfen, ob die Artefakt-ID innerhalb der Aufgabe bereits existiert und das entsprechende Objekt dann ersetzen bzw. das Artefakt hinzufügen, falls sie noch nicht vorhanden ist. Das Löschen bestehender Artefakte wäre nur durch eine Erweiterung der in der taskxml definierten Typen möglich. Allerdings ist diese Operation nicht zwangsläufig notwendig, da Artefakte hinzugefügt werden können. Anstatt es standardmäßig hinzuzufügen und bei Bedarf zu löschen, könnte es standardmäßig nicht vorhanden sein und bei Bedarf hinzugefügt werden. Dies sollte zumindest in vielen Fällen eine funktionierende Alternative sein, so auch bei der Beispielaufgabe. Entweder kann der Test standardmäßig vorhanden sein und entfernt werden, falls nicht die schwierige Variante ausgewählt wurde oder er kann standardmäßig nicht vorhanden sein und hinzugefügt werden, falls der Variationspunkt den Wert „hard“ hat.

Ein weiterer Ansatz, der an dieser Stelle nur kurz diskutiert, sonst aber nicht weiter verfolgt wird, da er zu diesem Zeitpunkt den Umfang der Arbeit übersteigen würde, wäre eine Art Plugin-System. Die Instanziierung durch den IS würde so erfolgen wie zuvor, lediglich die Instanziierung durch den Grader nicht. Der Grundgedanke ist, dass jede Aufgabe(nschablone) in gewisser Weise individuell ist. Aufgaben für einen bestimmten Grader haben Merkmale, die sie von Aufgaben für einen anderen Grader unterscheiden, aber untereinander möglicherweise gleich sind. Von diesen ähnlichen Aufgaben für denselben Grader haben einige möglicherweise noch weitere individuelle Merkmale, die sie von anderen ähnlichen Aufgaben unterscheiden. Der Grader, der für die Instanziierung zuständig ist, kann deswegen nicht auf konkrete Aufgaben eingehen, da hierfür der Source-Code für jede Aufgabe angepasst werden müsste. Um das zu umgehen, könnte eine Plugin-Schnittstelle definiert werden. Aufgabe eines Plugins wäre es, die Instanziierung durchzuführen. Somit könnte es ein Plugin geben, das grader-spezifische Merkmale instanziiert und weiterhin optional pro Aufgabe ein Plugin, das aufgaben-spezifische Merkmale bearbeitet. Beide Plugins könnten direkt in der task.zip enthalten sein, sodass es weiterhin ausreicht, mithilfe dieser eine vollständige Instanziierung durchzuführen. Die Instanziierung durch einen Grader würde entfallen und ersetzt werden durch die Instanziierung einer Komponente, die Plugins einer bestimmten Programmiersprache lädt und ausführt. Auf diese Weise müsste nicht für jeden Grader ein Instanziierungs-Backend implementiert werden, sondern lediglich pro Plugin-Sprache. Denn es ist irrelevant, was Gegenstand der Aufgabe ist und welcher Grader die Aufgabe letztlich bewertet, da ohnehin das Plugin oder die Plugins die Instanziierung übernehmen. Es ist nur relevant, in welcher Sprache die Plugins geschrieben sind, um sie laden und ausführen zu können.

Wenn also zum Beispiel die Plugins für einen Java-Grader, einen SQL-Grader und einen UML-Grader alle in Java geschrieben sind, kann für alle dasselbe Backend benutzt werden.
Bezogen auf die Beispielaufgabe könnte die Instanziierung wie folgt aussehen. Der Instanziierungsservice erhält eine Aufgabenschablone, die potentiell zwei Plugins (JAR-Dateien, falls sie in Java geschrieben sind) enthalten könnte: Eines für grader-spezifische Instanziierungen und eines für aufgaben-spezifische Instanziierungen. Das Beispiel hat nicht die nötige Komplexität um beiden Plugins eine Aufgabe zu geben, sodass in diesem konkreten Fall jeweils nur eines sinnvoll wäre. Zunächst führt der Instanziierungsservice seine normale Arbeit durch und ersetzt die Platzhalter. Anschließend übergibt er die Plugins an ein Backend-Plugin. Dieses Backend-Plugin fungiert als eine Art Adapter zwischen dem in Java geschriebenen Instanziierungsservice und der Sprache, in der die Plugins geschrieben sind (in diesem Fall zwar auch Java, denkbar wären aber auch andere Sprachen). Das Backend führt diese Plugins dann nacheinander aus, indem es ihnen die Aufgabenschablone und die Wertebelegung übergibt. Die Plugins instanziieren die Aufgabenschablone dann im Rahmen ihres Könnens und geben diese instanziierte Schablone zurück (noch keine fertige Instanz, da die Schablone ggf. noch weiter instanziiert wird). Im Beispiel könnte das grader-spezifische Plugin die Instanziierungsfunktion eines bestimmten Graders übernehmen und zum Beispiel wie oben beschrieben in den gradinghints nach einem Element *addTestDependingOnVpValue* suchen und entsprechend ggf. das Test-Element hinzufügen. Das aufgaben-spezifische Plugin hätte es einfacher, da es direkt auf die konkrete Aufgabe eingehen kann. Somit würde direkt im Source-Code stehen, dass geprüft werden soll, ob der Variationspunkt „difficulty“ den Wert „hard“ hat und im gegebenen Fall ein entsprechendes Test-Element hinzugefügt werden soll.
Der Vorteil liegt darin, dass auf einfache Weise aufgaben-spezifische Instanziierungen durchgeführt werden können. Für die Entwickler eines Graders, der variable Programmieraufgaben beinhalten soll, würde potentiell auch Arbeit entfallen, da sie „nur“ ein Plugin für Aufgaben ihres Graders entwickeln müssen und dabei möglicherweise auf ein bereits bestehendes System zurückgreifen können, das ihr Plugin ausführt. Falls dies nicht existiert, müsste ein „IS-Programmiersprachen-Backend-Plugin“, sowie das Backend zum Ausführen des Plugins in ihrer Programmiersprache entwickelt werden. Allerdings ist dies vermutlich ähnlich viel Arbeit, wie das Entwickeln eines IS-Grader-Backend-Plugins und einer Komponente für den Grader, die die grader-spezifische Instanziierung übernimmt. Grundsätzlich schließt sich dieses Konzept auch nicht mit dem Rest der Arbeit aus, da es den Instanziierungsservice nicht direkt betrifft. Solcherlei Plugins könnten durchaus in Aufgabenschablonen enthalten sein und über die IS-Grader-Backend-Plugin-Schnittstelle können nicht nur Grader angesprochen werden, sondern auch Komponenten, die mittels der Plugins eine Instanziierung durchführen. Semantisch würde das zwar nicht dem entsprechen, wofür die Schnittstelle eigentlich gedacht ist, aber denkbar und möglich wäre es.

5.5 Verarbeitung des Task Templates

Der Instanziierungsservice muss mit der in der Aufgabenschablone enthaltenen task.xml arbeiten. So kann es zum Beispiel, wie in den vorangegangenen Abschnitten verdeutlicht, notwendig sein, dass er eine oder mehrere Listen (mit Instanziierungsinformationen) verarbeiten, sowie Teile der oder auch die ganze task.xml instanziieren muss. Dies könnte jeweils direkt in der task.xml als Textdatei oder als String innerhalb von Java geschehen, indem zum Beispiel die Instanziierungsanweisungen geparst werden und die Instanziierung mit einfachem Suchen/Ersetzen auf den gesamten Inhalt passiert. Allerdings hat dieser Ansatz mehrere Probleme. So ist zum Beispiel nicht sichergestellt, dass die Aufgabenschablone überhaupt konform mit dem taskxml-Format ist, es ist schwierig erweiterbar, falls noch weitere Aktionen durchgeführt werden sollen und es ist auch nicht allzu robust gegen Änderungen. Gerade in einer objektorientierten Sprache wie Java, würde es sich anbieten, die task.xml mittels unmarshalling in eine Objektstruktur zu überführen, um mit dieser ganz normal arbeiten zu können. Insbesondere, da durch das XML-Schema eine strikte Struktur vorgegeben ist und es für den Prozess sehr gute und weit verbreitete Bibliotheken gibt. Auf diese Weise könnte sichergestellt werden, dass das taskxml-Format eingehalten wird, auf den Objekten könnten noch beliebige andere Operationen ausgeführt werden und bei Änderungen würde es in vielen Fällen ausreichen, die Objektstruktur bzw. das entsprechende Mapping anzupassen. Übrig bleibt noch die Frage, wann die Instanziierung der task.xml-Datei stattfinden soll - vor dem Unmarshalling oder danach. Davor hätte den Vorteil, dass alles mit einem Durchgang instanziiert werden könnte. Allerdings besteht die Gefahr, dass unbeabsichtigt Teile instanziiert werden, die hierfür nicht vorgesehen waren, da sie fälschlicherweise als Platzhalter interpretiert werden und im schlimmsten Fall die XML-Struktur ungültig wird. Nach dem Unmarshalling könnte die Objektstruktur durchwandert und die entsprechenden Attribute jeweils einzeln instanziiert werden. Dies bringt zwar etwas Overhead mit sich, allerdings ist davon auszugehen, dass dieser bei der voraussichtlichen Größe der Aufgaben(schablonen) so gering ausfällt, dass er vernachlässigt werden kann. Ein weiterer Vorteil wäre, dass feingranular und dynamisch vorgegeben werden könnte, welche Elemente instanziiert werden sollen und welche nicht, wenngleich dies zum aktuellen Zeitpunkt nicht geplant ist.

5.6 Sicherheitsaspekte

Wenngleich der Instanziierungsservice sehr wahrscheinlich in einer relativ sicheren Umgebung ausgeführt werden wird, sollten grundlegende Sicherheitsaspekte dennoch beachtet werden. So stellt sich zum Beispiel die Frage, ob der Service beliebige Anfragen entgegennimmt und bearbeitet oder hierfür eine Authentifizierung notwendig ist. Grundsätzlich kann mittels des IS kein großer Schaden angerichtet werden. Allerdings wäre es dennoch denkbar, dass der IS durch eine große Anzahl von Instanziierungsvorgängen annähernd unbrauchbar gemacht wird, falls beliebige Systeme eine Instanziierung anfordern können.

Weiterhin ist die Schnittstelle zur Anforderung grader-spezifischer Wertemengen potentiell kritisch, da hier Informationen abgefragt werden könnten, die nicht öffentlich einsehbar sein sollen. Dies könnte zum Beispiel passieren, wenn grader-spezifische Wertemengen aus einer Datenbank ausgelesen werden und die hierzu notwendige Abfrage direkt vom Client kommt und ohne weitere Überprüfungen ausgeführt wird.
Ein weiteres, potentielles Risiko, besteht in den weiteren Instanziierungsmöglichkeiten, wie sie in Abschnitt 3.4 schon kurz angesprochen wurden. Insbesondere das dort genannte Beispiel ausführbarer kleiner Skripte kann besonders kritisch sein, da hier potentiell ein großes Sicherheitsrisiko besteht. Bei dem Entwurf einer solchen Erweiterung sollte direkt von Beginn an bedacht werden, wie dieses Risiko möglichst gering gehalten werden kann. Allerdings gilt dies auch für andere Erweiterungen, da alle ihre eigenen Aufgaben und dafür benötigte Rechte haben und somit auch alle ein individuelles Risiko darstellen, das im Einzelfall abgewogen und bewertet werden sollte.

5.7 Anfragen an den Instanziierungsservice synchron oder asynchron

Die Anfragen an den Instanziierungsservice asynchron zu modellieren würde für den Fall Sinn machen, dass die Instanziierungsoperation lange dauert und der jeweilige Aufrufer somit nicht bis zur Antwort blockiert wird. Allerdings würde dies auch die Komplexität erhöhen, sowohl des Services als auch des Aufrufers. Die Instanziierung durch den IS ist keine komplexe Operation, die tendenziell eher schnell abgearbeitet werden kann. Die Instanziierung durch den Grader ist potentiell komplexer, die Laufzeit kann aber nicht allgemein bestimmt werden. Alternativ zur asynchronen Modellierung des Services, könnte der Aufrufer selbst auch dafür sorgen, dass er durch den Aufruf des synchronen Instanziierungsservices nicht komplett blockiert.

5.8 Zeitgleiche Bearbeitung mehrerer Anfragen

Die Instanziierung durch den IS ist eine Operation, die grundsätzlich ohne Probleme gleichzeitig für mehrere Aufgabenschablonen durchgeführt werden kann, da die hierfür notwendigen Daten alle in der Aufgabenschablone enthalten sind. Die Frage, ob eine parallele Bearbeitung Sinn macht, hängt somit zum einen vom System ab, auf dem der IS läuft, ob dieses parallele Bearbeitung ermöglicht, was in der Regel wohl der Fall sein wird und zum anderen davon, wie das Grader-Backend mit mehreren nahezu zeitgleichen Anfragen umgeht. Denn selbst wenn der IS zum Beispiel acht Instanziierungen echt zeitgleich bearbeiten kann, der Grader alle Anfragen aber sequentiell abarbeitet, wird der Performance-Gewinn eher gering ausfallen. Der Unterschied zu einer sequentiellen Abarbeitung durch den IS wäre, dass teilinstanziierte Schablonen direkt in die Warteschlange des Graders eingereiht werden und dieser somit permanent am Arbeiten ist. Würde der

IS ebenfalls sequentiell arbeiten, müsste er die Antwort des Graders erst verarbeiten, anschließend mit der nächsten Instanziierung beginnen und sobald er mit dieser von seiner Seite aus fertig ist, die Schablone an den Grader übergeben, sodass dieser immer kurze Pausen hätte, in denen er auf Arbeit wartet.

5.9 Schwerwiegende Fehler beim Instanziierungsservice

Wenngleich dieser Fall konzeptionell nicht auftreten sollte, wäre es dennoch denkbar, dass schwerwiegende Fehler entstehen, von denen der IS sich nicht erholen kann oder aufgrund derer er sogar komplett abstürzt. Der Instanziierungsservice kann mit diesem Fall unterschiedlich umgehen. Entweder er versucht die gerade bearbeitete, sowie sich möglicherweise in der Warteschlange befindende Anfragen wiederherzustellen, oder er versucht nichts dergleichen, startet einfach neu und wartet auf eingehende Anfragen.
Falls die Anfragen ohnehin asynchron bearbeitet werden, ist eine Wiederherstellung vergleichsweise einfach möglich, da Anfragen beim Eintreffen zum Beispiel persistiert werden können und der Aufrufer eine ID erhält, mit der er den Status der Bearbeitung abfragen kann. Die Löschung der Anfragen erfolgt erst dann, wenn der Aufrufer die Instanz tatsächlich abgerufen hat. Sollte der IS abstürzen, kann er einfach die nächste Anfrage aus der Warteschlange laden und seine Arbeit fortsetzen.
Bei einer synchronen Bearbeitung ist die Wiederherstellung eher schwierig. Die angefragten Schablonen könnten zwar zwischengespeichert werden, allerdings geht bei einem Absturz serverseitig die Verbindung zum Client verloren. Denkbar wäre, dass der Aufrufer bei einer Anfrage Daten mitsendet, mithilfe derer der IS sich im Fehlerfall wieder zum Aufrufer verbinden kann. Allerdings bringt das weitere Probleme mit sich, zum Beispiel, dass der Aufrufer netzwerktechnisch von außen erreichbar sein muss. Weiterhin lässt sich die damit verbundene Komplexität und der dadurch entstehende Overhead wohl nicht rechtfertigen, für einen Fall, der vermutlich sehr selten auftreten wird. Eine weitere Alternative wäre die Verwendung einer Art Session-System. Bei der ersten Anfrage könnte der Aufrufer eine Session-ID erhalten. Serverseitig wird alles so ausgeführt, dass der Instanziierungsprozess durch die Session-ID identifiziert werden kann. Sollte es beim IS zu einem größeren Problem und damit zu einem Verbindungsabbruch kommen, könnte der Aufrufer versuchen, sich mithilfe der Session-ID wieder neu zu verbinden, ohne einen ganz neuen Instanziierungsprozess anzustoßen. Denkbar wäre in diesem Fall auch, alle notwendigen Informationen unter der Session-ID persistent abzulegen, sodass der IS selbst nach einem Absturz die Anfragen laden und weiter bearbeiten kann. In diesem Fall müssten allerdings auch noch weitere Fragen geklärt werden - wann zum Beispiel mit der Bearbeitung fortgesetzt werden soll oder was passiert, falls ein Aufrufer nicht versucht, sich neu zu verbinden? Unabhängig davon, bestünde implizit allerdings schon eine Lösung. Denn für den Aufrufer stellt sich die Situation nur so dar, dass der IS ihm nicht antwortet. Dies könnte auch an ganz anderen Problemen, wie Netzwerkfehlern oder einer Überlastung des IS liegen. Für diese Fälle wird der Aufrufer sehr wahrscheinlich ohnehin eine Art Timeout setzen, nach Ablauf dessen die Anfrage neu gestartet oder dem

Benutzer ein Fehler präsentiert wird. Somit kann der IS ebenfalls einfach neugestartet werden und auf neue Anfragen warten.

5.10 Gleichzeitige Anbindung mehrerer Backends

Es wäre denkbar, dass gleichzeitig mehrere Grader-Backends an den Instanziierungsservice angeschlossen sind. Auf diese Weise müsste pro System nur ein Instanziierungsservice laufen, der alle Anfragen empfängt, teilverarbeitet und sie anschließend an das richtige Backend weiterleitet. Die Alternative dazu wäre, jeweils nur ein Grader-Backend pro IS-Instanz zu erlauben, sodass ggf. mehrere Instanzen gestartet werden müssten. Einerseits wäre es praktisch, dass alle Instanziierungsanfragen an denselben Service gerichtet werden können und es wäre weniger Adminstrationsaufwand. Andererseits wäre es möglicherweise sauberer, die Anfragen für verschiedene Grader voneinander zu trennen. Weiterhin hätte man bei mehreren gleichzeitigen Backends eine Art „Single point of failure“. Tritt bei einzelnen Backends ein Fehler auf, würde dies nur den Instanziierungsservice für diesen Grader betreffen. Ein weiterer Punkt ist, dass mehrere Backends natürlich ebenfalls die Komplexität der Anwendung erhöhen würden.

6 Lösung

Im folgenden Kapitel wird die Umsetzung des Instanziierungsservices näher erläutert. Hierfür werden zunächst die dafür neu entwickelten Modelle vorgestellt und anschließend erklärt, wie der Instanziierungsservice mithilfe dieser die Instanziierung durchführt. Anschließend wird noch einmal genauer auf einzelne Aspekte der Implementierung eingegangen.

6.1 Model-Übersicht

6.1.1 InstantiationInstructions

Die InstantiationInstructions dienen dem Instanziierungsservice und ggf. auch dem Grader-Backend als Anleitung dafür, wie genau sie die Instanziierung durchzuführen haben. In Abbildung 6.1 sind sie als UML-Klassendiagramm dargestellt. Das „Wurzelobjekt" InstantiationInstructions existiert nur einmal und enthält alle weiteren Anweisungen.
Zunächst hat es eine Liste von ActionParametern, die als Standardparameter für alle durchzuführenden Instanziierungen gültig sind und bei Bedarf pro Artefakt und Aktion überschrieben werden können. Auf diese Weise kann die Angabe immer gleicher Parameter verringert werden. Falls zum Beispiel alle Dateien dieselbe Codierung besitzen, muss dies nur einmal als Standard-Parameter angegeben werden, anstatt es bei jedem Datei-Artefakt zu wiederholen. Momentan gibt es drei unterstützte Parameter-Typen: Placeholder, FileParameter und AdvancedParameter. Placeholder dienen dazu, das Platzhalter-Format für Variationspunkte dynamisch pro Artefakt angeben zu können. Hiermit wird dem in Abschnitt 5.3 beschriebenen Problem Rechnung getragen, dass ein allgemeingültiges Platzhalterformat zu Problemen führen könnte, falls es in einem der Inhalte der Artefakte gleichzeitig auch in einem anderen Kontext und nicht als Platzhalter enthalten ist. Mithilfe des FileParameters wird das im selben Abschnitt beschriebene Problem gelöst, die Codierung pro Artefakt angeben zu können. Der AdvancedParameter dient im Grunde als eine Art Fallback für alle beliebigen Parameter, die eine beliebige grader instantiation engine für ihre eigene Instanziierung benötigt, wie es in Abschnitt 5.4 beschrieben wurde. Der Instanziierungsservice selbst bedient sich zum gegenwärtigen Zeitpunkt nur der Placeholder und FileParameter.
Weiterhin enthalten die InstantiationInstructions noch TemplateArtifacts und TaskActions. Ein TemplateArtifact referenziert ein in der task.xml-Datei enthaltenes Artefakt, das in irgendeiner Weise instanziiert werden soll, wobei die Referenzierung mittels der

artifactType- und artifact_id-Attribute erfolgt. Ein TemplateArtifact kann grundsätzlich beliebig viele, muss aber mindestens eine, ArtifactAction besitzen. Erst die ArtifactActions geben an, wie genau das Artefakt instanziiert werden soll. Aktuell gibt es nur die einfache Suchen-und-Ersetzen-Instanziierung durch den Instanziierungsservice selbst oder die advanced-Instanziierung durch eine grader instantiation engine. Außerdem kann jede ArtifactAction ActionParameter haben, die die Standard-Parameter überschreiben. In der jetzigen Implementierung sind die verschiedenen ArtifactActions lediglich als enum realisiert. Denkbar wäre, diese Unterscheidung zukünftig als Subklassen von ArtifactAction zu gestalten. Dies hätte den Vorteil, dass action-spezifische Attribute direkt im entsprechenden Element bzw. der entsprechenden (Sub-)Klasse untergebracht werden könnten. Für die beiden momentan verwendeten Action-Typen ist das allerdings nicht notwendig, sodass dies zunächst nicht umgesetzt wurde, da es den Entwurf zusätzlich verkomplizieren würde.
TaskActions dienen dazu, Task-weite Instanziierungen zu veranlassen, die sich nicht auf ein spezielles Artefakt beziehen.
Sie können aktuell entweder den Typ „simple_search_and_replace“ oder „advanced“ haben. Der erste Fall sagt dem Instanziierungsservice, dass er eine task-weite Instanziierung durchführen soll, indem er alle enthaltenen Artefakte instanziiert. Ausnahme hiervon sind file-Artefakte - diese müssen immer explizit als TemplateArtifact aufgelistet sein, um instanziiert zu werden. TaskActions besitzen ebenfalls ActionParameter, die die Standard-Parameter überschrieben. Bei der task-weiten Instanziierung wird für jedes Artefakt vor der Instanziierung überprüft, ob es ein TemplateArtifact gibt, dass dieses Artefakt referenziert. Wenn dem so ist und das TemplateArtifact eine Suchen-und-Ersetzen-ArtifactAction besitzt, überschreiben dessen Parameter die TaskAction- und die Standard-Parameter. Weiterhin bedeutet das, dass file-Artefakte durch die Task-weite Instanziierung ebenfalls instanziiert werden, falls ein TemplateArtifact mit entsprechender ArtifactAction exisitiert, die dieses file-Artefakt referenziert. Eine TaskAction vom Typ „advanced“ sagt dem IS, dass die Aufgabenschablone auch noch an die grader instantiation engine weitergegeben werden soll.
Die „simple_search_and_replace“-TaskAction gibt an, dass alle Artefakte instanziiert werden soll. Standardmäßig bedeutet das, dass der Instanziierungsservice sie durch einfaches Suchen-und-Ersetzen der Platzhalter instanziiert. Gibt es zu einem Artefakt aber gleichzeitig auch ein TemplateArtifact, überschreibt dieses die vorzunehmenden Aktionen, da es spezifischer als die TaskAction ist. Enthält dieses TemplateArtifact dann z.B. nur eine advanced-TaskAction, wird dieses Artefakt auch nur durch die grader instantiation engine instanziiert und nicht durch einfaches Suchen-und-Ersetzen. Soll das ebenfalls passieren, muss diese ArtifactAction auch noch einmal separat im TemplateArtifact aufgeführt werden.

Eine Möglichkeit, die InstantiationInstructions für die Beispielaufgabe umzusetzen, ist in Listing 6.1 aufgeführt. Zunächst wird der allgemeine Platzhalter-Parameter angegeben, der für die ganze Schablone gilt. Anschließend wird dem Instanziierungsservice über die simple_search_and_replace Task-Action mitgeteilt, dass alle Artefakte durch ihn instanziiert werden sollen. Die zweite TaskAction, vom Typ advanced, gibt einer fikti-

ven grader instantiation engine über ein eigenes Element *addTestDependingOnVp* den Auftrag, den enthaltenen Test hinzuzufügen, wenn der angegebene Variationspunkt den angegebenen Wert hat.

Wenngleich es momentan sowohl für TaskActions, als auch für ArtifactActions, nur zwei mögliche Typen von Instanziierungen gibt (Suchen/Ersetzen und advanced), ist dieses Modell dennoch sehr gut erweiterbar, um auch zukünftige Instanziierungsmöglichkeiten zu unterstützen. So kann die Liste der als „action_type“ erlaubten Aktionen beliebig erweitert werden, um auch andere Instanziierungen zu unterstützen, ebenso können beliebige neue Parameter-Typen definiert werden.
Eine solche mögliche Erweiterung ist in Listing 6.2 dargestellt. Hier existiert eine fiktive Komponente, die artifact_actions des Typs „execution“ verarbeitet. Diese beinhalten ein Skript innerhalb eines Skript-Parameters. Die Komponente führt das Skript aus und übergibt diesem alle Attribute des Artefaktes, zu dem die action gehört und weiterhin die Wertebelegung. In diesem Beispiel prüft das Skript, ob zwei Variationspunkte einen bestimmten Wert haben und verändert die referenzierte Datei der model-solution, zu der die action gehört. Anschließend gibt es die (veränderten) Attribute zurück und die Komponente aktualisiert die Attribute der model-solution mit den zurückgegebenen Werten.
Wie genau solche weiteren Instanziierungsmöglichkeiten in den Instanziierungsservice integriert werden können, wird in Abschnitt 6.5 näher erläutert.

```
<vpt:instantiation_instructions
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:vpt='urn:hshannover:proforma:extension:
   ↪ variableprogrammingtask:v1.0'
    xmlns:p="urn:proforma:task:v1.1"
    xmlns:x="grader:xy:namespace">

    <vpt:default_parameter xsi:type="vpt:placeholder" prefix="%
       ↪ vp{" suffix="}"/>

    <vpt:task_action id="simple_root" action_type="
       ↪ simple_search_and_replace"/>

    <vpt:task_action id="addHardTest" action_type="advanced">
        <vpt:action_parameter xsi:type="vpt:advanced_parameter">
            <x:addTestDependingOnVp vpName="difficulty" vpValue=
               ↪ "hard">
                <p:test id="testHard">
                    <p:title>Negative input test</p:title>
                    <p:test-type>unittest</p:test-type>
                    <p:test-configuration xmlns:x="grader:xy:
                       ↪ namespace">
                        <x:testMethod>testNegativeInputFibonacci
                           ↪ </x:testMethod>
                    </p:test-configuration>
                </p:test>
            </x:addTestDependingOnVp>
        </vpt:action_parameter>
    </vpt:task_action>

</vpt:instantiation_instructions>
```

Listing 6.1: XML-Instanzdokument der InstantiationInstructions für die Beispielaufgabe

```
<vpt:instantiation_instructions
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:vpt='urn:hshannover:proforma:extension:
   ↪ variableprogrammingtask:v1.0'>

     <vpt:template_artifact id="01" artifact_id="model-solution
        ↪ -01" type="model-solution">
        <vpt:artifact_action id="updateMs1Fileref" action_type="
           ↪ execution">
           <vpt:action_parameter xsi:type="vpt:script_parameter"
              ↪ >
              function(attributes, values){
                if(values.x === 3 || values.y === "easy"){
                  attributes.filerefs.fileref[0].refid="file1
                     ↪ ";
                }
                return attributes;
              }
           </vpt:action_parameter>
        </vpt:artifact_action>
     </vpt:template_artifact>

</vpt:instantiation_instructions>
```

Listing 6.2: Mögliche Erweiterung der InstantiationInstructions

6.1.2 GraderInstancedArtifacts

Die in Abbildung 6.2 dargestellten GraderInstancedArtifacts zeigen das Format der Antwort der grader instantiation engine an den Instanziierungsservice als UML-Klassendiagramm. Darin enthalten sind die instanziierten Artefakte, die vom IS noch mit in die Schablone integriert werden sollen. Mit einer Ausnahme sind alle Attribute der GraderInstancedArtifacts Elemente, die ebenfalls in der task.xml-Datei benutzt werden, um wie in Abschnitt 5.4 beschrieben die Typsicherheit direkt an der Schnittstelle sicherzustellen und Mapping-Operationen o.ä. überflüssig zu machen. Die genannte Ausnahme sind die InstancedFileContents, die den Inhalt einer Datei darstellen, die in einer task.zip-Datei enthalten ist. Hierfür war ein neues Element bzw. eine neue Klasse notwendig, da dies im ProFormaA-Format nicht abgedeckt wird. Das File-Element verweist im Falle einer „externen“, nicht eingebetteten, Datei nur auf diese, kann aber nicht deren Inhalt aufnehmen. Grundsätzlich könnte man zwar fordern, dass nur eingebettete Dateien benutzt werden dürfen, allerdings sollte die Erweiterung eines bestehenden Formats dieses nicht einschränken, indem bestehende Elemente verboten bzw. nicht unterstützt werden. Weiterhin wäre es dann nicht direkt möglich Binärdateien zu verwenden, da der Inhalt des file-Elements vom Typ String ist. Somit müsste der Binärinhalt z.B. manuell base64

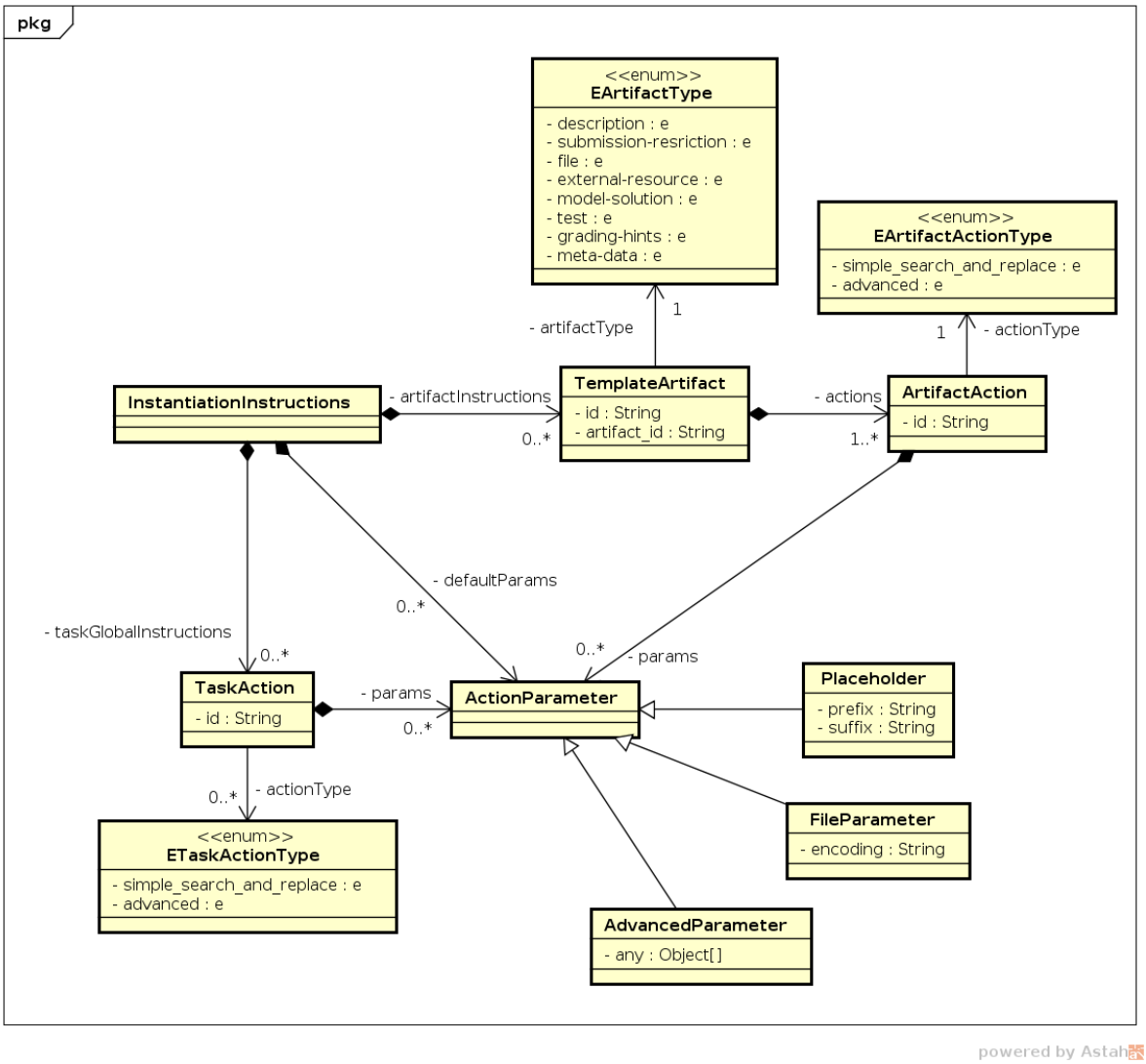

Abbildung 6.1: Klassenmodell der InstantiationInstructions

codiert/decodiert und in einem eingebetteten file-Element abgelegt werden. Dies ist aber umständlich und entspricht semantisch auch nicht der dem Standard nach empfohlenen Verwendung von eingebetteten Dateien.
FileRef im InstancedFileContents-Element ist ein String, der als Referenz die ID eines File-Elements enthält, das dann die weiteren Informationen, wie zum Beispiel den Dateinamen, bereithält. Soll eine Datei der Aufgabenschablone hinzugefügt werden, müssen somit immer zwei Artefakte zurückgegeben werden: Ein InstancedFileContent-Element, das den Inhalt der Datei enthält und ein File-Element, das die weiteren Informationen vorhält und auf das das InstancedFileContent-Element verweist.
Falls instanziierte Artefakte zurückgegeben werden, werden diese, je nach Typ, jeweils unterschiedlich in die Aufgabenschablone integriert:

- Description und Submission-Restrictions ersetzen die hierfür bereits vorhandenen Elemente, da sie jeweils nur einmal existieren
- Grading-Hints ersetzen nur das any-Element, da VariabilityInfo und Instantiation-Instructions ohnehin nicht instanziiert werden können, aber weiterhin benötigt werden (siehe auch Abschnitt 6.1.4)
- Das Meta-Data-Element wird ebenfalls komplett ersetzt, da die enthaltenen any-Elemente nicht sinnvoll miteinander abgeglichen werden können, da sie beliebige Elemente enthalten können
- Files, External-Resources, Model-Solutions und Tests werden jeweils der Liste bereits vorhandener Artefakte des entsprechenden Typs hinzugefügt. Falls die ID eines instanziierten Artefaktes mit der ID eines bereits vorhandenen Artefaktes desselben Typs übereinstimmt, ersetzt das instanziierte Artefakt das bereits vorhandene
- InstancedFileContents ersetzen ebenfalls den Inhalt bereits vorhandener Dateien oder geben den Inhalt neu hinzugefügter Dateien an.

In der Beispielaufgabe würden die GraderInstancedArtifacts dafür genutzt werden, das von der grader instantiation engine ggf. erzeugte Test-Element an den Instanziierungsservice zurückzugeben, damit dieser es der Aufgabenschablone hinzufügen kann und der entsprechende Test beim Bewerten durchgeführt wird. Ein Beispiel, wie diese XML-Rückgabe aussehen könnte, ist in Listing 6.3 aufgeführt.

```
<vpt:grader_instanced_artifacts
    xmlns:vpt='urn:hshannover:proforma:extension:
   ↪ variableprogrammingtask:v1.0'
    xmlns:p="urn:proforma:task:v1.1">
   ↪
    <vpt:instanced_tests>
        <p:test id="testHard">
            <p:title>Negative input test</p:title>
            <p:test-type>unittest</p:test-type>
            <p:test-configuration xmlns:x="grader:xy:namespace">
                <x:testMethod>testNegativeInput</x:testMethod>
            </p:test-configuration>
        </p:test>
    </vpt:instanced_tests>
</vpt:grader_instanced_artifacts>
```

Listing 6.3: XML-Instanzdokument der GraderInstancedArtifacts für die Beispielaufgabe

6.1.3 Transferobjekte

Für den Aufruf des Instanziierungsservices bzw. die Übergabe der Aufrufparameter sowie der Rückgabe der Aufgabeninstanz, werden Transferobjekte benötigt. Dies hat zwei Gründe: Zum einen müssen beim Aufrufen mehrere Parameter übertragen werden, nämlich die Aufgabenschablone und die Wertbelegung. JAX-RS erlaubt für POST-Funktionen allerdings nur einen Parameter, weshalb ein Container-Objekt benötigt wird. Zum anderen können wie in Abschnitt 6.1.2 bereits beschrieben, weitere in der task.zip-Datei enthaltene Dateien neben der task.xml nicht im ProFormA-Format übertragen werden, weshalb auch hier ein Transfer-Objekt benötigt wird. Abbildung 6.3 zeigt die verwendeten Transfer-Objekte als UML-Klassenmodell.
InstantiationRequestTO ist das Container Objekt, das für die Anfrage notwendig ist und die Aufgabenschablone, sowie die Wertbelegung enthält. Letztere ist eine Klasse aus der in Abschnitt 6.1.5 beschriebenen Bibliothek. Die Aufgabenschablone selbst ist noch ein weiteres Transfer-Objekt, das Datei-Objekte enthält, die nur aus Dateiname und Dateiinhalt bestehen. Hier wird also nicht auf das Wissen zurückgegriffen, dass in der task.zip-Datei eine Aufgabenschablone enthalten ist, sondern es werden alle enthaltenen Dateien gleichberechtigt als einzelne FileTO-Objekte übertragen. Eine andere Möglichkeit wäre gewesen, die Aufgabenschablone als Zip-Datei als einzelne FileTO zu übertragen. Es wurde sich allerdings für den anderen Ansatz entschieden, da der Instanziierungsservice sonst mit der Zip-Datei umgehen müsste und in den meisten Fällen der Aufrufer die Zip-Datei vermutlich ohnehin bereits entpackt hat, um eine Wertebelegung zu generieren. Sollte sich dennoch die Notwendigkeit ergeben, eine ganze Zip-Datei an den IS übergeben zu können, kann die entsprechende Unterstützung an der Eingangsschnittstelle leicht nachgepflegt werden.

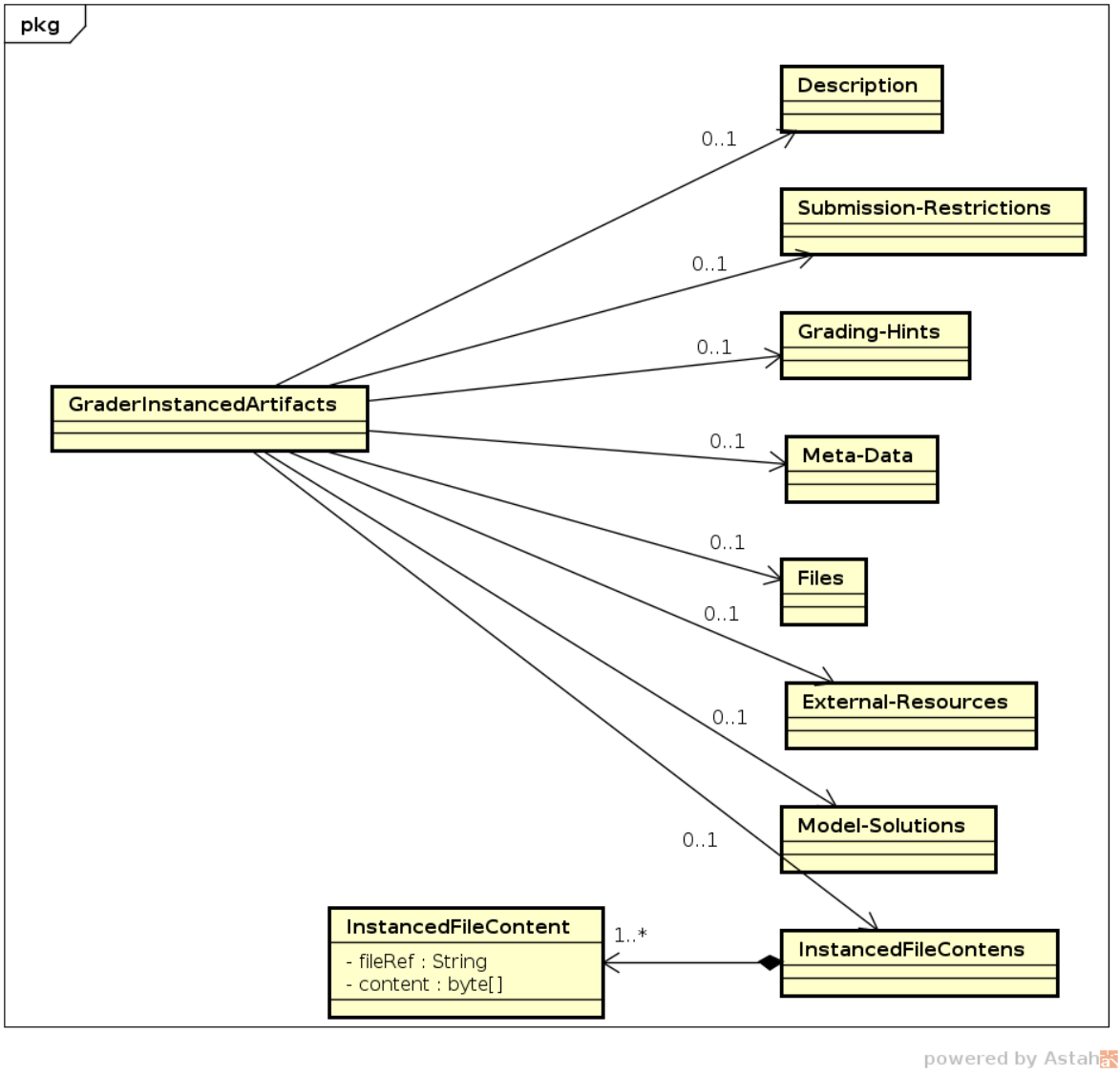

Abbildung 6.2: Klassenmodell der GraderInstancedArtifacts

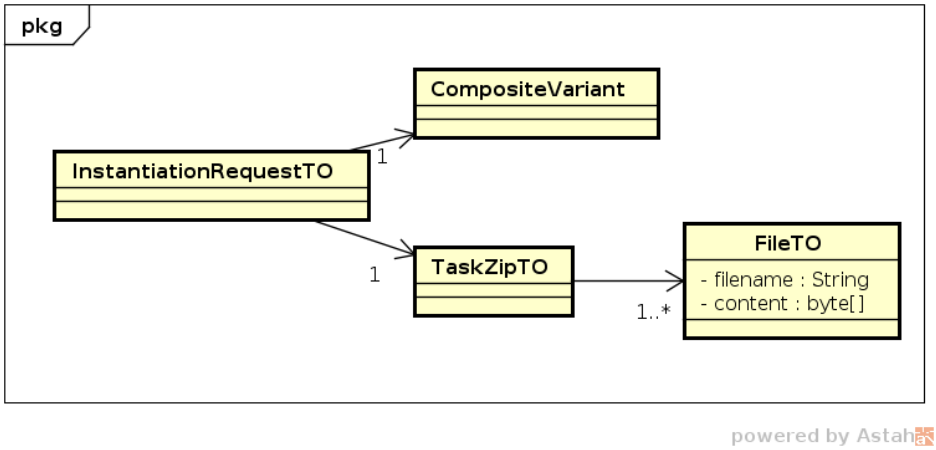

Abbildung 6.3: Klassenmodell des Transfer-Objekts

6.1.4 Integration in die Aufgabenschablone

Die Spezifikation der Variabilität sowie die InstantiationInstructions müssen in irgendeiner Weise in die Aufgabenschablone integriert werden. Denkbar wäre dies zum Beispiel als eigene Datei(en) in der task.zip-Datei oder aber direkt innerhalb der task.xml-Datei. Erstere Option wäre zwar durchaus möglich, allerdings ist eines der Ziele des ProFormA-Aufgabenformats, innerhalb eines Formats alle relevanten Informationen über die Aufgabe selbst zu vereinen. Vor diesem Hintergrund macht es mehr Sinn, die entsprechenden Elemente mit in die Aufgabe zu integrieren. Da das Format aber keine Variabilität vorsieht, gibt es auch kein Element, in das die beiden Elemente direkt hineinpassen würden. Somit könnte entweder ein neues, optionales Kind-Element des Task-Elements („variability“ o.ä.) eingeführt werden oder sie könnten im Rahmen einer der Erweiterungspunkte, also dem any-Element von grading-hints oder meta-data, aufgenommen werden. Ein eigenes variability-Element hätte den Vorteil, dass es einen semantisch sinnvollen Platz gäbe. Allerdings würde man damit sehr wahrscheinlich die Unterstützung anderer Tools und Programme für ProFormA-Aufgaben brechen, da es nicht dem Standard entspricht. Somit ist es zunächst sinnvoller, das grading-hints- oder das meta-data-Element hierfür zu nutzen. Da weder die Spezifikation der Variabilität, noch die InstantiationInstructions Meta-Daten sind, wurde sich für das grading-hints-Element entschieden. Somit enthalten Aufgabenschablonen im grading-hints-Element immer die InstantiationInstructions und die VariabilityInfo. Letztere enthält für Aufgabenschablonen die Spezifikation der Variabilität und für Aufgabeninstanzen die Angabe, welcher Variationspunkt welchen konkreten Wert hat. Längerfristig ist es allerdings anzustreben, ein variability-Element in die Spezifikation des Aufgabenformates mit aufzunehmen - dies hätte den weiteren Vorteil, dass andere Tools und Programme, die das Format unterstützen, explizit Funktionen für die Variabilitäts-Informationen zur Verfügung stellen könnten. Ebenso würden bei einer Validierung der Aufgabenschablone die Variabilitäts-Elemente mit validiert werden; beim momentan gewählten Ansatz muss manuell überprüft werden, ob die entsprechenden

Elemente vorhanden sind.

6.1.5 libvts

Für alle Aufgaben, die sich auf die in [3] beschriebene Variabilität beziehen, wurde die von Prof. Dr. Robert Garmann hierzu entwickelte „libvts“-Bibliothek verwendet. Beispiele hierfür sind die Übertragung der Wertebelegung, die in Abschnitt 6.1.4 beschriebene VariabilityInfo, sowie das Generieren einer zufälligen Wertebelegung im Client-Stub, der in Abschnitt 6.9 noch genauer erläutert wird.

6.1.6 Erweiterung der Models um Mappings

In Abschnitt 6.1.1 wurde bereits erwähnt, dass die TemplateArtifacts in den InstantiationInstructions Artefakte in der task.xml-Datei referenzieren. Diese Referenz wird beim Unmarshalling allerdings nicht in eine echte Objektreferenz überführt, da InstantiationInstructions und die task.xml prinzipiell auch in zwei unterschiedlichen Dateien enthalten sein könnten. Die Referenzierung besteht somit weiterhin nur insofern, als dass die „artifact_id“ der „id“ eines Artefaktes entspricht. Wie in Abschnitt 6.2.2 noch deutlich werden wird, ist es während des Instanziierungsprozesses notwendig, diese Referenzen aufzulösen und zum Beispiel einen Test mit einer bestimmten id zu finden. Die von XJC[1] zum ProFormA-Aufgabenformat standardmäßig generierten Klassen enthalten aber nur normale Listen von Artefakten, sodass es notwendig wäre, für jedes zu suchende Artefakt die Liste des entsprechenden Typs so lange linear zu durchsuchen, bis das Objekt mit der entsprechenden id gefunden wird. Insbesondere bei vielen zu instanziierenden Artefakten, muss dieser Vorgang häufig wiederholt werden. Aus diesem Grund wurde in allen Model-Klassen, die eine solche Liste von Artefakten enthalten, neben der Liste auch noch eine HashMap hinzugefügt, die als Schlüssel die ID des Artefakts hat und als Wert das Artefakt selbst. Somit können Zugriffe in nahezu konstanter Zeit realisiert werden. Für wenige zu instanziierende Artefakte bringt dies keinen Performancegewinn, da die HashMaps zunächst einmal initialisiert werden müssen, bei einer größeren Anzahl an Artefakten überwiegt aber die Ersparnis. Um das Initialisieren für jedes Objekt nicht manuell anstoßen zu müssen und auch offen für Erweiterungen zu sein, wurde ein Interface mit nur einer parameterlosen setup-Methode eingeführt. Alle Klassen, die nach dem Unmarshalling noch etwas zu initialisieren haben, implementieren dieses Interface. Beim Unmarshaller wird ein eigener Listener registriert, der nach dem Unmarshalling jedes Objektes aufgerufen wird und prüft, ob das Objekt das Interface implementiert. Falls es das tut, wird die entsprechende Interface-Methode aufgerufen.

[1] Tool, das annotierte Java-Klassen zu einem gegebenen XML-Schema generiert (Oracle Docs XJC)

6.2 Logik-Übersicht

Im folgenden Abschnitt wird auf den Kern des Instanziierungsservices eingegangen. Hierfür wird zunächst anhand eines Klassendiagramms erklärt, welche Klassen daran beteiligt sind, wie sie zueinander stehen und welche Verantwortlichkeiten sie haben und anschließend wird mittels eines Sequenzdiagramms der Ablauf einer Instanziierung näher erläutert.

6.2.1 Verantwortlichkeiten

In Abbildung 6.4 sind die Kernklassen des Instanziierungsservices abgebildet, die für die eigentliche Instanziierung zuständig sind. Weitere (Helfer)klassen, die nur indirekt daran beteiligt sind, sind nicht aufgeführt.
TaskInstanceResource ist die Schnittstelle des Instanziierungsservices an die Außenwelt. Mittels eines POST-Requests an die Adresse „*<server>*/taskinstance“ kann der IS angesprochen werden. Jede Anfrage wird vom application server automatisch in einem eigenen Thread ausgeführt, sodass auch die nachfolgende Instanziierung parallel zu anderen Instanziierungen ablaufen kann, wie es bereits in Abschnitt 5.8 besprochen wurde. Die Klasse nimmt lediglich Requests entgegen, führt das Mapping zwischen den technischen Transferobjekten und fachlichen Modellen durch und beauftragt den rein fachlichen InstantiationManager mit der eigentlichen Instanziierung. Dieser leitet und überwacht den Instanziierungsprozess, indem er einige allgemeine Operationen selbst ausführt und sich verschiedener anderer Klassen für die restliche Instanziierung bedient. Eine davon ist der SimpleInstantiationHandler, der für die Suchen-und-Ersetzen-Instanziierung zuständig ist. Er selbst kümmert sich allerdings nur darum, welche Parameter für das aktuelle Artefakt gültig sind und welche Platzhalter durch welche Werte ersetzt werden sollen. Für die eigentliche Instanziierung nutzt er eine Klasse, die das SimpleInstancer-Interface implementiert, welches Methoden für die Instanziierung unterschiedlicher Datentypen anbietet. Wieso es hier noch einmal eine Aufteilung gibt, wird in Abschnitt 6.4 genauer erläutert. Falls notwendig ruft der InstantiationManager nach der einfachen Instanziierung noch die grader instantiation engine auf. Hierfür existiert das GraderInstantiationEnginePlugin-Interface, das von einer entsprechenden Klassen implementiert werden muss. Diese Klasse ist die einzige, für die während der Instanziierung nicht eine eigene Instanz erzeugt wird, sondern dieselbe Instanz in allen Instanziierungsprozessen verwendet wird. Weiterhin existiert noch das SimpleInstantiable-Interface, das alle Klassen implementieren, die vom Instanziierungsservice durch einfaches Suchen-und-Ersetzen instanziierbar sein sollen. Die interface-Methode bekommt ein SimpleInstancer-Objekt, mithilfe dessen das Artefakt seine Variablen selbst instanziieren kann, siehe hierzu auch Abschnitt 6.3.

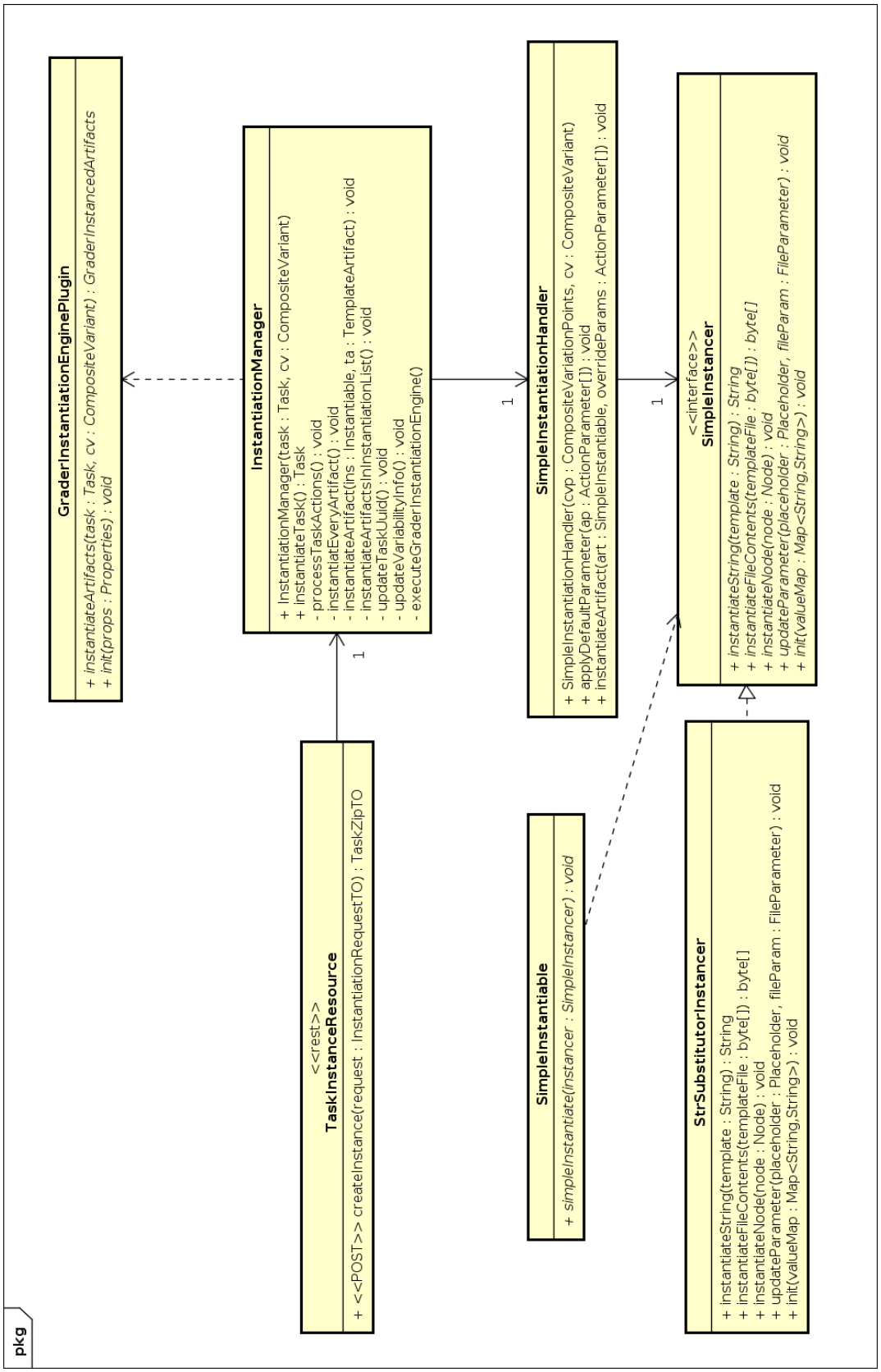

Abbildung 6.4: Klassenmodell des Kerns des Instanziierungsservices

6.2.2 Ablauf

Abbildung 6.5 zeigt den Ablauf während einer Instanziierung als Sequenzdiagramm. Dargestellt ist lediglich die Instanziierung an sich - die Initialisierung der einzelnen beteiligten Objekte, sowie die Verwendung weiterer Hilfsklassen o.ä. wurde zwecks Lesbarkeit weggelassen. Ein Diagramm inklusive Initialisierung ist im elektronischen Anhang enthalten.[2]
Anfragen an den Instanziierungsservice werden, wie in Abschnitt 5.7 diskutiert, synchron ausgeführt, da erste Tests gezeigt haben, dass die Dauer einer Instanziierung hierfür nicht zu lang ist (vgl. Testergebnisse in Abschnitt 7.2). Der Instanziierungsservice wurde als Webanwendung entwickelt und benötigt somit einen Web-Container, in dem er ausgeführt wird. Für diese Arbeit war hierfür Apache Tomcat[3] festgelegt. Bei einer Anfrage erstellt der Container eine Instanz der Klasse TaskInstanceResource und ruft deren createTask-Methode mit dem Transfer-Objekt als Parameter auf. Die Methode erstellt aus dem Transfer-Objekt ein Aufgabenschablonen-Objekt und das Wertebelegungs-Objekt. Um sicherzustellen, dass die übergebene Aufgabenschablone dem ProFormA-Aufgabenformat entspricht, wird die task.xml-Datei während des Unmarshallings validiert. Dies schließt eine Validierung der InstantiationInstructions mit ein. In Abschnitt 3.4 wurde beschrieben, dass Artefaktschablonen vor ihrer Instanziierung möglicherweise ungültig sein können. Eine Validierung an der Eingangsschnittstelle würde dies aber nicht zulassen. Zunächst wurde sich der Typsicherheit halber dazu entschieden, nur valide Aufgabenschablonen zuzulassen. Werte, die im Rahmen der Instanziierung erst gesetzt werden sollen, können in der Aufgabenschablone entsprechend den Standardwerten, die in der Variabilitätsspezifikation enthalten sind, belegt werden. Sollte es sich später als notwendig herausstellen, auch ungültige Aufgabenschablonen anzunehmen, kann die Validierung der task.xml unabhängig von der Validierung der InstantiationInstructions einfach entfernt werden. Weiterhin erstellt die Methode einen neuen InstantiationManager und übergibt ihm die beiden erstellten Objekte (Aufgabenschablone und Wertbelegung). Der Instantiation-Manager prüft, ob die Aufgabenschablone die wichtigsten Objekte enthält und erstellt anschließend eine neue SimpleInstantiationHandler-Instanz, die die Wertebelegung und eine Liste der Variationspunkte erhält. Diese wiederum erstellt eine SimpleInstancer-Instanz und initialisiert diese mit der Information, welcher Variationspunkt welchen Wert hat. Zum Schluss werden dem SimpleInstantiationHandler-Objekt noch die Standard-Parameter übergeben.

Nach der Initialisierung aller Objekte beginnt der in der Abbildung dargestellte Teil der Instanziierung. Zunächst werden in 1.1 die TaskActions verarbeitet - für die „advanced“-Action wird einfach ein Flag gesetzt, dass später noch die grader instantiation engine aufgerufen werden soll und für die „simple_search_and_replace“-Action wird ein Flag gesetzt, dass alle Artefakte instanziiert werden sollen. Zusätzlich werden die Parameter dieser TaskAction noch an den SimpleInstantiationHandler übergeben, da diese die

[2] Diagramm „Instantiation-Complete“ in der Datei doc/uml/ConceptIS.asta
[3] Website des Apache Tomcat Projektes

Standardparameter überschreiben.

Je nachdem, ob das Flag zur Instanziierung aller Parameter gesetzt wurde, wird entweder wie in der Abbildung unter 1.2 dargestellt die Methode instantiateEveryArtifact aufgerufen oder, falls das Flag nicht gesetzt ist, die instantiateArtifactsInInstantiationList-Methode. Im ersten Fall werden nach und nach alle Artefakte durchlaufen und überprüft, ob es zu diesem Artefakt gleichzeitig auch noch ein TemplateArtifact gibt, das noch genauere Anweisungen zur Instanziierung beinhalten würde. Existiert ein solches, wird wie in 1.2.1 die instantiateArtifact-Methode aufgerufen, die als Parameter das Artefakt und das TemplateArtifact erhält. Existiert es nicht, erhält die Methode als zweiten Parameter null.
Die Instanziierung des Description-Artefakts in der Beispielaufgabe würde in diesem Fall so ablaufen, dass die Description in jedem Fall instanziiert wird, da das Flag zur Instanziierung aller Artefakte gesetzt ist. Es wird überprüft, ob es zu der Description auch noch ein Template-Artifact gibt, da die darin enthaltenen ArtifactActions und Parameter spezifischer sind, als die TaskAction zur Instanziierung aller Artefakte. Anschließend wird die instantiateArtifact-Methode mit der Description als Parameter aufgerufen (1.2.1). Der zweite Parameter ist entweder das TemplateArtifact, falls es existiert oder null, falls es nicht existiert.

Falls das Flag zur Instanziierung aller Artefakte nicht gesetzt ist, werden alle TemplateArtifacts durchlaufen, das Artefakt herausgesucht, das sie referenzieren und beides ebenfalls an die instantiateArtifact-Methode übergeben. Von dort an ist der Ablauf für beide Varianten (Flag gesetzt/Flag nicht gesetzt) wieder gleich.
In der Beispielaufgabe würde in diesem Fall keine simple_search_and_replace TaskAction vorhanden sein, sondern eine Menge von TemplateArtifacts, die jedes zu instanziierende Artefakt einzeln angeben. Unter anderem würde ein solches für die Description existieren, das wiederum eine ArtifactAction vom Typ simple_search_and_replace hat, die ihrerseits einen überschreibenden Platzhalter-Parameter enthalten könnte. Das Description-Objekt würde herausgesucht und zusammen mit dem TemplateArtifact-Objekt an die instantiateArtifact-Methode übergeben werden.

Die Methode prüft zunächst, ob ein TemplateArtifact mit übergeben wurde. Ist das nicht der Fall, aber das Flag zur Instanziierung aller Artefakte ist gesetzt und das Artefakt ist außerdem vom Typ SimpleInstantiable, wird es dem SimpleInstantiationHandler ohne artefakt-spezifische Überschreibungs-Parameter zur Instanziierung übergeben. Dieser Fall tritt immer dann auf, wenn durch die simple_search_and_replace-TaskAction angegeben wurde, dass alle Artefakte instanziiert werden sollen, zu dem gerade bearbeiteten Artefakt aber kein TemplateArtifact angegeben wurde. Somit wird dieses Artefakt dann durch Suchen/Ersetzen mittels der allgemeingültigen Parameter instanziiert.

Wurde aber ein TemplateArtifact mitgegeben, werden die darin enthaltenen ArtifactActions ausgeführt. Für „advanced“-Actions wird wiederum nur das Flag für die grader instantiation engine gesetzt. Bei simple_search_and_replace-Actions wird das Artefakt zu SimpleInstantiable gecasted und wie in 1.2.1.1 dargestellt an den SimpleInstantiationHandler übergeben, zusammen mit den ggf. vorhandenen artefakt-spezifischen Überschreibungs-Parametern der Action.
In der Beispielaufgabe könnte es sein, dass für die Description ein eigenes Platzhalter-Format gewählt und somit hierfür ein TemplateArtifact erstellt wurde, um dieses angeben zu können. Das TemplateArtifact würde eine simple_search_and_replace-ArtifactAction beinhalten, die wiederum einen Platzhalter-Parameter beinhaltet. Die Methode würde die Description dann zu einem SimpleInstantiable casten und zusammen mit dem Platzhalter-Parameter an den SimpleInstantiationHandler übergeben (1.2.1.1).

Der SimpleInstantiationHandler gibt seinem SimpleInstancer-Objekt zunächst die für dieses Artefakt geltenden Parameter (1.2.1.1.1) und ruft anschließend auf dem SimpleInstantiable (dem übergebenen Artefakt), die simpleInstantiate-Methode mit dem SimpleInstancer-Objekt als Parameter auf (1.2.1.1.2). Das Artefakt kümmert sich dann selbst um seine Instanziierung (1.2.1.1.2.1 beispielhaft für einen String), indem es mittels des SimpleInstancers seine Attribute instanziiert, näheres dazu in Abschnitt 6.3.
Im Beispiel würde der SimpleInstantiationHandler dem SimpleInstancer-Objekt den Platzhalter-Parameter übergeben (1.2.1.1.1) und anschließend die simpleInstantiate-Methode der Description mit dem SimpleInstancer-Objekt als Parameter aufrufen (1.2.1.1.2). Die Description würde dieses Objekt benutzen, um ihr description-Attribut zu instanziieren (1.2.1.1.2.1).

Nachdem alle Artefakte durch den Instanziierungsservice instanziiert wurden (Ende von 1.2), wird, abhängig davon, ob das entsprechende Flag gesetzt wurde, die grader instantiation engine aufgerufen (1.3) und dabei die bereits teil-instanziierte Aufgabenschablone, sowie die Wertebelegung übergeben. Die zurückgegebenen, instanziierten Artefakte, werden wie in Abschnitt 6.1.2 bereits erläutert, in die Aufgabenschablone integriert.
Anschließend ersetzt der InstantiationManager noch das template-Objekt in der VariabilityInfo in den Grading-Hints durch ein instance-Objekt, sodass dort nicht mehr die Spezifikation der Variabilität enthalten ist, sondern die Angabe, welchem Variationspunkt in dieser Instanz welcher Wert zugewiesen wurde (1.4).
Zum Schluss generiert der InstantiationManager noch eine neue Version 5 UUID für die Aufgabeninstanz (1.5). Hierbei wird als namespace die UUID der Aufgabenschablone benutzt und als name die Konkatenation der Werte der Variationspunkte, getrennt durch ein Semikolon. Die UUID der Aufgabeninstanz wird auf die so neu generierte UUID gesetzt und die parent-uuid auf die UUID der Aufgabenschablone.
Die TaskInstanceResource-Klasse mappt die Aufgabeninstanz daraufhin wieder auf das entsprechende Transfer-Objekt und gibt dieses als Antwort auf die Anfrage zurück. Während des Marshallings der Aufgabeninstanz, wird diese gleichzeitig validiert. Dadurch

wird sichergestellt, dass Aufrufer des Instanziierungsservices sich darauf verlassen können, dass sie eine ProFormA-kompatible Aufgabe erhalten.

6.3 Instanziierung im Model

Die Model-Klassen sind selbst für ihre Instanziierung zuständig, indem sie per Method-Injection ein Instanziierungs-Objekt übergeben bekommen, mithilfe dessen sie ihre Attribute instanziieren können. Die Alternative zu diesem Ansatz wäre gewesen, die Model-Klassen als reine Datenhaltungs-Klassen zu betrachten und einen separaten Controller o.ä. zu implementieren, der sowohl das Artefakt, als auch das Instanziierungs-Objekt kennt und das Artefakt „von außen“ instanziiert. Auf diese Weise hätte man Logik und Datenhaltung noch weiter entkoppelt und bei größeren Änderungen am Model, bei denen die entsprechenden Java-Klassen ggf. auch neu generiert werden, müsste der Instanziierungs-Code nicht neu implementiert oder auf die neuen Klassen übertragen werden. Allerdings ist das ProFormA-Aufgabenformat mittlerweile in einem Stadium, in dem große Änderungen sehr unwahrscheinlich sind. Für den Fall, dass dies doch noch einmal passieren sollte, ist es möglicherweise sogar besser, dass der entsprechende (kurze) Instanziierungscode noch einmal neu implementiert werden muss - denn wenn größere Änderungen am Model vorgenommen werden, ist die Wahrscheinlichkeit hoch, dass dies auch die Instanziierung betrifft. Wird bestehender Code dann einfach nur angepasst, kann es passieren, dass notwendige Änderungen übersehen werden und zum Beispiel ein Attribut weiterhin instanziiert wird, obwohl es in der neuen Version nicht mehr instanziiert werden sollte. Weiterhin ist der Grundgedanke dieses Ansatzes, dass das Model selbst am besten weiß, welche seiner Attribute instanziiert werden sollen. Bei kleinen Änderungen am Model muss nur die entsprechende Klasse angepasst werden, da die Instanziierung mit in dieser Klasse liegt.
Eine Besonderheit stellt allerdings das Instanziieren von XML-any-Elementen dar, die in einigen Artefakten enthalten sind. Da dies beliebige Objekte sein können, über die der Instanziierungsservice kein Wissen besitzt, kann nicht auf einzelne Attribute o.ä. eingegangen werden. Aus diesem Grund werden bei diesen Elementen, wenn sie im Rahmen des sie enthaltendenen Artefaktes instanziiert werden sollen, sowohl alle Attribute, als auch der Textinhalt, instanziiert. Dies könnte grundsätzlich zu einem Problem führen, falls der Textinhalt z.B. Base64 codiert ist und somit nicht instanziiert werden sollte. Praktisch ist es aber sehr unwahrscheinlich, dass es dort zu Problemen kommt, da das Platzhalter-Format in den meisten Fällen vermutlich mittels Sonderzeichen dargestellt wird und in Base64 nur die Sonderzeichen /, + und = erlaubt sind. Weiterhin kann dieses Problem einfach umgangen werden, indem das Platzhalterformat für das entsprechende Artefakt spezifisch angepasst wird.

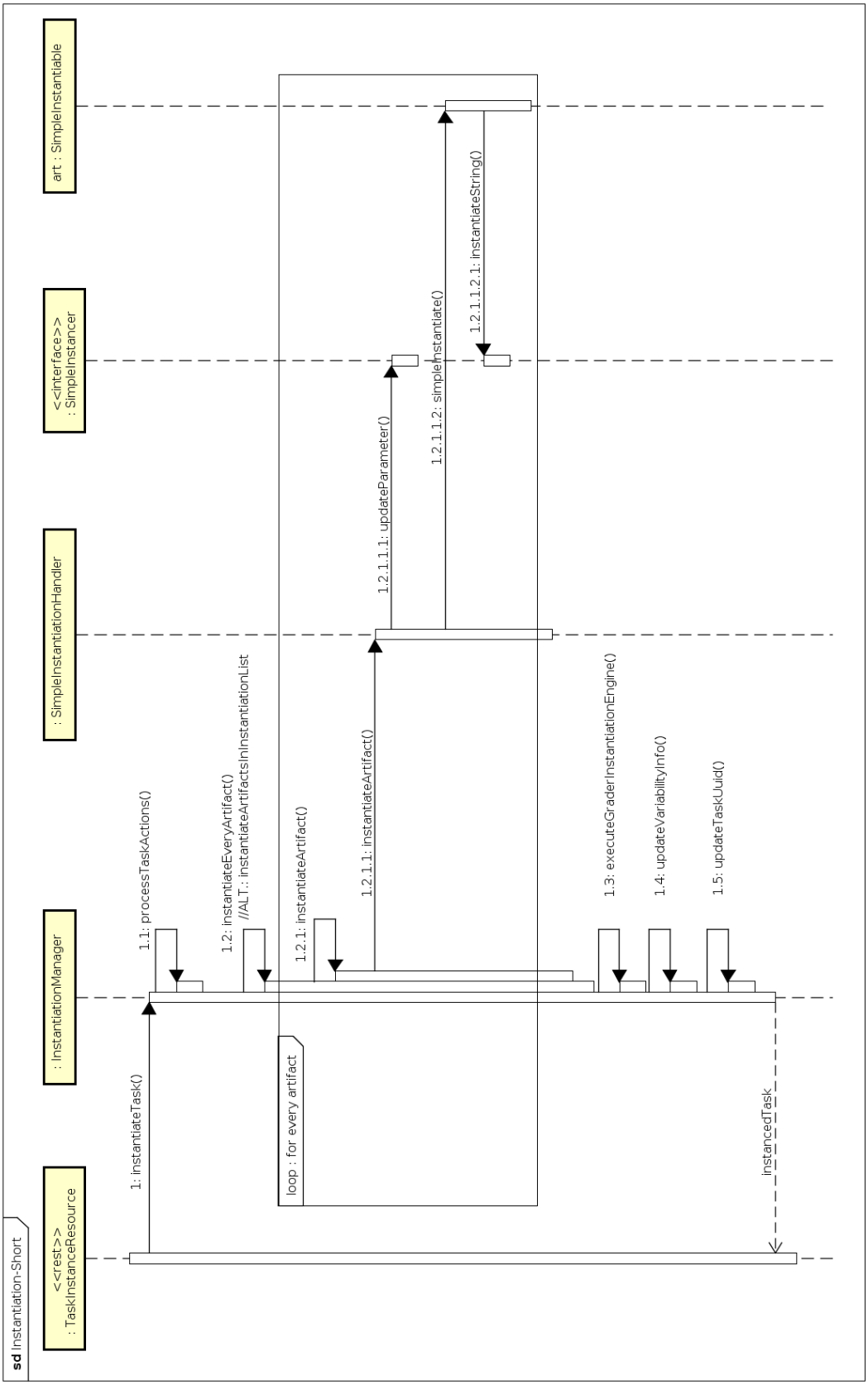

Abbildung 6.5: Sequenzdiagramm des Instanziierungsprozesses

6.4 Einfache Instanziierung

Es stellt sich die Frage, warum ein extra Handler existiert, der wiederum eine Referenz auf ein Objekt besitzt, das ein bestimmtes Interface implementieren muss, anstatt eine einzige Klasse zu entwerfen, die für die gesamte Suchen-und-Ersetzen-Instanziierung zuständig ist. In Abschnitt 5.3 wurde angesprochen, dass geklärt werden muss, ob eine bereits bestehende Template-Engine verwendet oder diese Funktionalität selbst implementiert wird. Die Recherche hierzu hat ergeben, dass es viele sehr gute Template-Engines mit weitreichenden Funktionalitäten gibt. Da der Großteil dieser Funktionen aktuell nicht benötigt wird, es aber durchaus denkbar ist, sie später einmal zu verwenden, wurde das SimpleInstancer-Interface entsprechend dem Adapter-Pattern eingeführt. Auf diese Weise können beliebig komplexe Template-Engines verwendet werden, ohne dass der Client-Code, der sie benutzt, angepasst werden muss. So wäre es zum Beispiel möglich anstatt des einfachen Suchen/Ersetzens eine bereits bestehende Template-Sprache zu verwenden, die Kontrollstrukturen wie if/else, Schleifen oder ähnliches enthält, indem eine die Sprache unterstützende Bibliothek eingebunden und ein Adapter dafür implementiert wird, der das SimpleInstancer-Interface implementiert. Es wäre sogar denkbar, mehrere Template-Engines gleichzeitig zu unterstützen und dynamisch in den InstantiationInstructions angeben zu können, welche Implementierung verwendet werden soll. Zusammen mit dem SimpleInstantiationHandler entspräche dies dem Strategy-Pattern, welches in vielen Systemen gängige Praxis ist.
Der SimpleInstantiationHandler wurde eingeführt, da es IS-spezifische fachliche Anforderungen gibt, die jeder SimpleInstancer umsetzen muss, so zum Beispiel das Verwalten der aktuell gültigen Parameter. Um den dafür notwendigen Code nicht redundant implementieren zu müssen, übernimmt diese Aufgaben der Handler. Eine Alternative wäre gewesen, SimpleInstancer als abstrakte Klasse zu modellieren und den allgemeinen Code dort zu implementieren, entsprechend dem Template-Pattern. Allerdings ist dieser Ansatz weniger flexibel. Weiterhin besteht mit dem gewählten Ansatz eine klare Trennung zwischen dem für die Instanziierung notwendigen „Verwaltungs-Code“ und dem eigentlichen Instanziierungs-Code, sodass es klar definierte Verantwortlichkeiten gibt und der Adapter auch wirklich nur für das Adaptieren einer Template-Engine verantwortlich ist.
Die momentan verwendete „Template-Engine“ ist die StringSubstitutor-Klasse[4] aus dem *apache.commons.text*-package. Sie bietet genau die erforderlichen Funktionalitäten (Prefix und Suffix des Platzhalter-Formats definieren und Platzhalter ersetzen) und ist sehr einfach einzubinden und zu initialisieren. Weitere unterstützte Features sind das rekursive Ersetzen von Platzhaltern (z.B. %vp{%vp{a}} zu %vp{b} zu c, falls der Platzhalter a den Wert „b“ und Platzhalter b den Wert „c“ hat), sowie das Escapen von Platzhaltern. So wird \%vp{class} z.B. nicht durch „Person“ ersetzt (falls der Platzhalter class den Wert „Person“ hat), sondern es wird nur das Escape-Zeichen entfernt, sodass daraus %vp{class} würde. Diese Template-Engine wurde gewählt, da sie alle Anforderungen erfüllt, gleichzeitig aber auch sehr schlank ist und ein einfaches Setup besitzt. Andere Template-Engines mit teils deutlich größerem Funktionsumfang haben das „Problem“,

[4] Apache Docs

dass das Setup entsprechend der Funktionalität ebenfalls komplexer ist und man sich damit insgesamt unnötige Komplexität in die Anwendung holen würde, die eigentlich gar nicht notwendig wäre, da der größte Teil des Funktionsumfangs dieser Template-Engines momentan ohnehin nicht benötigt wird.
Eine Einschränkung, die die einfache Instanziierung in der implementierten Lösung hat ist die, dass numerische Attribute nicht mit Platzhaltern versehen werden können. Dies liegt daran, dass die Attribute der entsprechenden Modell-Klassen aufgrund der Typsicherheit ebenfalls numerisch sind und somit keine Platzhalter aufnehmen können. Verhindert wird dies auch durch die in Abschnitt 6.2.2 angesprochene Validierung an der Eingangsschnittstelle. Diese Einschränkung ist in der Praxis allerdings weniger relevant, da es nur sehr wenige numerische Attribute gibt und diese zu denen gehören, die tendenziell vermutlich eher selten zwischen Aufgabeninstanzen variieren. Weiterhin bedeutet das nicht, dass sie gar nicht instanziiert werden können - eine grader instantiation engine, die direkt mit den numerischen Attributen arbeitet, könnte diese Instanziierung sehr wohl durchführen. Ein in der Praxis größeres Problem ist, dass auf diese Weise auch Referenzen auf IDs und Attribute mit einer fest vorgegebenen Wertemenge im Allgemeinen keine Variationspunkte enthalten dürfen. Dies würde seitens der Implementierung der Modell-Klassen zwar zugelassen werden, wird allerdings durch die Validierung verhindert. Sollten sich diese Einschränkungen in der Praxis als zu groß herausstellen, könnte der Instanziierungsservice aber relativ einfach dahingehend angepasst werden, dass auch bei diesen Attributen Platzhalter erlaubt sind.

6.5 Erweiterbarkeit für weitere Instanziierungsmöglichkeiten

Momentan unterstützt der Instanziierungsservice nur das einfache Suchen/Ersetzen, wenngleich, wie in Abschnitt 6.4 bereits angesprochen, auch andere, mächtigere Template-Sprachen auf einfache Weise unterstützt werden können. Allerdings können nicht nur weitere Template-Engines relativ einfach hinzugefügt werden, sondern auch ganz andere Instanziierungsmechanismen, wie es im Laufe der Arbeit schon mehrfach angedeutet wurde. Dies könnte umgesetzt werden, indem ein neues Handler-Interface eingeführt wird, das im Grunde dem SimpleInstantiationHandler entspricht. Dieser und alle anderen InstantiationHandler würden dieses Interface dann implementieren. Somit ergäbe sich eine Art zweistufiges Adapter-Pattern, indem der InstantiationManager eine Referenz auf ein Objekt des Typs InstantiationHandler hätte, das allgemein einen bestimmten Instanziierungsmechanismus verwaltet und seinerseits eine Referenz auf ein Objekt mit einer konkreten Implementierung dieses Mechanismus besitzt. In Abbildung 6.6 ist dies beispielhaft dargestellt. Der Übersicht halber wurden Methoden weggelassen. Als weiterer Mechanismus wurde in diesem Beispiel ein ExecutionInstantiationHandler eingeführt, der in den Artefakten bzw. in den TemplateArtifacts oder deren Actions enthaltene kleine Skripte ausführen kann, in diesem Fall entweder in Javascript oder in Python. Dass dies

vom Modell der InstantiationInstructions unterstützt wird, bzw. das Modell dahingehend sehr einfach erweiterbar ist, wurde in Abschnitt 6.1.1 bereits angesprochen. Auch ein solches Beispiel wurde mit Listing 6.2 bereits eingeführt. Der InstantiationManager würde beliebig viele Handler kennen und kann beispielsweise über den action_type eines TemplateArtifacts entscheiden, an welchen Handler er das Artefakt zur Instanziierung übergibt. Gegebenenfalls könnte dann noch ein weiteres, optionales Attribut mit einem Standardwert eingeführt werden, anhand dessen der Handler wiederum entscheidet, an welche konkrete Implementierung er es übergeben möchte.
Solche Erweiterungen können grundsätzlich auf zwei verschiedene Arten passieren: Entweder sie werden direkt im Instanziierungsservice selbst implementiert, dies wäre die einfache Variante oder sie können dynamisch als eine Art Plugin hinzugefügt werden. Im zweiten Fall könnte ein neuer InstantiationHandler, zusammen mit einer oder mehreren konkreten Implementierungen entwickelt und in ein JAR-Archiv gebunden werden. Die JAR-Archive können vom Instanziierungsservice beim Starten geladen werden und wären somit verfügbar. Die Assoziation eines action_types zu einem bestimmten Handler könnte auf mehreren Wegen geschehen: Durch eine zusätzliche Angabe im TemplateArtifact, wenngleich das sehr aufwändig und redundant wäre, durch eine Angabe in der config-Datei des Instanziierungsservices oder zum Beispiel durch das Erstellen einer eigenen Annotation, die alle Handler hinzufügen müssen und die angibt, für welchen action_type dieser Handler gedacht ist. Über diese Annotation könnten beim Starten auf einfache Weise alle Handler gefunden werden. Dieselbe Methode könnte auch für konkrete Implementierungen verwendet werden - wenn es bereits einen Handler gibt, dieser aber nicht die konkrete Implementierung unterstützt, die benötigt wird, könnte dies selbst implementiert werden. Falls es zum Beispiel schon den ExecutionInstantiationHandler gibt, dieser aber noch keine konkrete Implementierung für PHP-Skripte bietet, könnte diese selbst imlpementiert werden. Der Instanziierungsservice kann beim Starten alle entsprechend annotierten Implementierungen suchen und sie dem jeweiligen Handler hinzufügen, ähnlich dem Dependency-Injection-Konzept, damit nicht alle Handler dies redundant implementieren müssen. Somit könnten von verschiedenen Entwicklern unterschiedliche Instanziierungsmechanismen modular hinzugefügt werden, die bei Bedarf auch noch von anderen Entwicklern in ihren Möglichkeiten erweitert werden können.

6.6 Sicherheitsaspekte

Wie in Abschnitt 5.6 diskutiert, wurde eine einfache Authentifizierung implementiert. Entsprechend der Anforderung, bei der Implementierung möglichst nahe an Grappa zu bleiben, wird hierfür die HTTP Basic Authentication verwendet. Bei jeder Anfrage an den Instanziierungsservice müssen somit Benutzername und Passwort mitgesendet werden.
Die korrekten Anmeldedaten können in der Konfigurationsdatei des Instanziierungsservices gesetzt werden. Momentan ist nur eine Benutzername/Passwort-Kombination

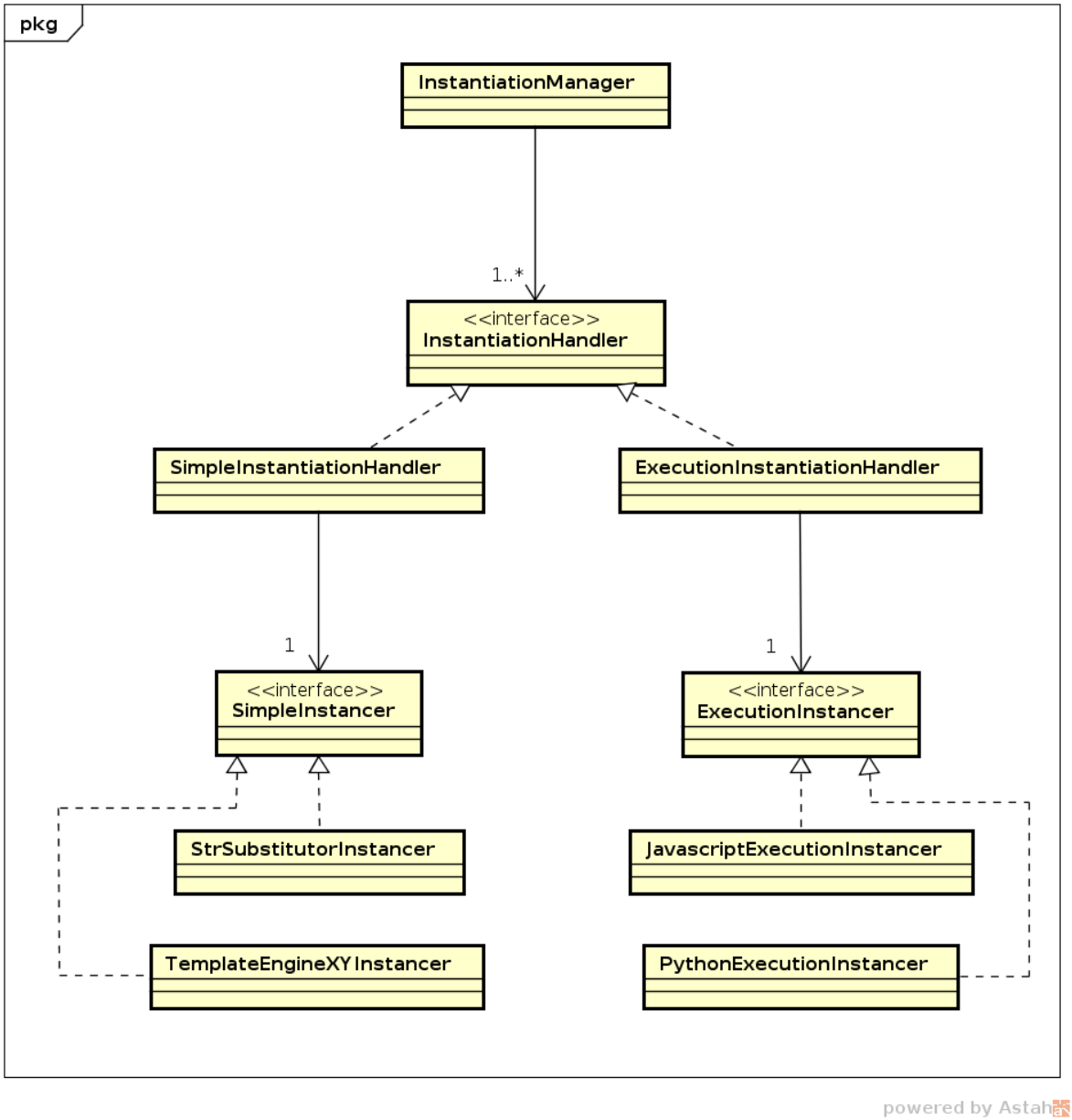

Abbildung 6.6: Vereinfachtes Klassenmodell mit mehreren InstantiationHandlern

möglich, sodass alle Nutzer des Instanziierungsservices dieselben Anmeldetaten verwenden. Unterschiedliche Anmeldedaten wären nötig, falls zum Beispiel unterschiedliche Nutzer unterschiedliche Recht haben, das ist hier aber nicht der Fall. Ein weiteres Anwendungsszenario wäre, dass unterschiedliche Systeme auf den Instanziierungsservice zugreifen und einem der Systeme die entsprechenden Rechte dafür entzogen werden sollen. Da der IS zunächst aber als Prototyp fungieren soll und ohnehin nur ausgewählte Systeme darauf zugreifen, ist auch das nicht zutreffend. Somit wurde diese Funktion mangels konkreten Nutzens zunächst nicht implementiert.
Wie schon in Abschnitt 5.6 angesprochen, müssen in Zukunft etwaige weitere Instanziierungsmöglichkeiten, wie sie auch in Abschnitt 6.5 noch weiter ausgeführt wurden, ebenfalls in das Sicherheitskonzept integriert werden. So sollten Skripte zum Beispiel nur Auswirkungen auf Aufgabenschablonen haben dürfen und möglichst isoliert vom Rest der Anwendung ausgeführt werden. Auch wären sie ein weiteres Argument dafür, mehrere Zugänge einzurichten, da über einen kompromittierten Zugang mithilfe der Skripte möglicherweise durchaus Schaden angerichtet werden könnte.
Auch das Nachladen „beliebiger“ InstantiationHandler und konkreter Instanziierungsimplementierungen, wie in 6.5 beschrieben, die mittels bestimmter Annotationen automatisch geladen und ausgeführt werden, stellt potentiell ein großes Sicherheitsrisiko dar. So wäre es denkbar, dass jemand mit Zugriff auf das Dateisystem des Servers, auf dem der Instanziierungsservice ausgeführt wird, dort schadhafte Plugins platziert, die beim nächsten Neustart automatisch geladen werden. Hat die Person gleichzeitig auch noch die Zugangsdaten, um eine Instanziierung anzustoßen, könnte somit das schadhafte Plugin angesteuert und auf diese Weise im Rahmen der laufenden JVM beliebige Befehle ausgeführt werden. Natürlich ist die erste Abwehrmaßnahme in diesem Fall die richtige und sichere Konfiguration des Servers und der darauf laufenden Anwendungen, gleichzeitig zeigt das Beispiel aber auch, dass sich durchaus auch im Rahmen des Sicherheitskonzeptes des Instanziierungsservices über solche Probleme Gedanken gemacht werden sollte, falls er dahingehend erweitert wird.

6.7 Grader-Instantiation-Engine-Plugin

6.7.1 Grader instantiation engine interface

Das Interface, das grader instantiation engines implementieren müssen, um vom Instanziierungsservice aufgerufen werden zu können, ist das GraderInstantiationEnginePlugin-Interface. Dieses hat zum einen eine init-Methode, der ein Properties-Objekt übergeben wird und zum anderen die instantiateArtifacts-Methode, die als Parameter die vom Instanziierungsservice schon teil-instanziierte Aufgabenschablone, sowie die Wertebelegung erhält und ein Objekt des in Abschnitt 6.1.2 beschriebenen Typs GraderInstancedArtifacts zurückliefert.
In der Konfigurationsdatei des Instanziierungsservices kann auf eine weitere Konfigurationsdatei für die grader instantiation engine verwiesen werden. Falls eine solche, nicht

leere Konfigurationsdatei angegeben wird, werden die darin enthaltenen Einstellungen beim Starten des Webservices als Properties-Objekt an die init-Methode übergeben. Wird keine Datei angegeben oder ist diese leer, wird die Methode gar nicht aufgerufen. Die instantiateArtifacts-Methode bekommt zwar die Aufgabenschablone übergeben, sollte diese aber als read-only ansehen. Sie wird benötigt, um eine vollständige Instanziierung zu ermöglichen, ist konzeptionell aber ein konstanter Wert, da die grader instantiation engine potentiell auf einem anderen Server liegt und somit ansonsten ein veränderbares Objekt über das Netzwerk übertragen werden müsste, was den Rahmen dieser Arbeit übersteigt. Aus diesem Grund wurden die GraderInstancedArtifacts als Rückgabewert entworfen. Sollte das GIE-Plugin dennoch unerlaubterweise Änderungen an der Schablone vornehmen, sind die Auswirkungen und Ergebnisse undefiniert. Im Idealfall sollte dieses Veränderungsverbot direkt im Code erzwungen werden, allerdings ist das in diesem Fall nicht sinnvoll möglich. Prinzipiell sollte der Parameter ein const-Parameter sein, wie er z.B. aus C++ bekannt ist. Ein solches Konzept kennt Java allerdings nicht. Eine andere Möglichkeit wäre, extra für diesen Fall eine zweite, unveränderbare („immutable“) Klassen-Struktur der Aufgabenschablone zu erstellen, allerdings ist das zu aufwändig. Die letzte Möglichkeit wäre, die Aufgabenschablone zu kopieren und nur eine Kopie an die grader instantiation engine zu geben, sodass etwaige Änderungen daran nicht auf das Original-Objekt zurückfallen. Allerdings ist dies für die potentiell recht großen Objektstrukturen von Aufgaben(schablonen) zu laufzeitintensiv.
Falls bei der Instanziierung ein schwerwiegender Fehler auftritt, kann die Methode eine GraderInstantiationEngineException werfen.
Im Gegensatz zu allen anderen Instanziierungs-Klassen ist diese die einzige, von der nicht ein neues Objekt für jede Instanziierung erstellt, sondern dieselbe Instanz für alle benutzt wird. Somit dürfen dort keine nicht-thread-sicheren Operationen durchgeführt werden.

6.7.2 Grader instantiation engine stub

Der Stub arbeitet auf sehr einfache Weise: Er überprüft, ob eine Aufgabenschablone eine TaskAction vom Typ „advanced“ besitzt und weiterhin, ob diese TaskAction einen AdvancedParameter hat. Anschließend durchläuft er alle Objekte, die in dem Parameter enthalten sind und sucht nach ihm bekannten Objekttypen, wie zum Beispiel Description, File oder ModelSolution. Also genau die, die auch in der Antwort in den GraderInstancedArtifacts enthalten sein können. Diese Objekte fügt er dann der Antwort hinzu. Somit können alle beliebigen Szenarien getestet werden, da in der Aufgabenschablone dynamisch angegeben werden kann, was die Rückgabe des grader instantiation engine plugins sein soll.

Ein etwas längeres Beispiel, wie die InstantiationInstructions für den stub aussehen könnten, ist in Listing 6.4 aufgeführt. Da es sich in diesem Fall um einen Ausschnitt aus einer Aufgabenschablone handelt, fehlen an dieser Stelle die Angaben der jeweiligen Namespaces. Das InstantiationInstructions-Element enthält eine TaskAction vom Typ advanced, die wiederum einen Parameter vom Typ advanced enthält. In diesem befinden

sich drei ProFormA-Elemente. Eine external-resource, eine model-solution und ein Test. Diese würden nun von dem stub ausgelesen und der Antwort hinzugefügt werden, sodass die entsprechend zurückgegebenen GraderInstancedArtifacts genau diese drei Elemente beinhalten würden.

Diese Verwendung der advanced Actions ist allerdings sehr speziell auf die stub-Natur der Komponente zugeschnitten. Eine echte grader instantiation engine würde sehr wahrscheinlich nicht einfach vorgefertigte Elemente aus den advanced-Parametern auslesen und diese der Antwort hinzufügen, sondern viel häufiger Manipulationen an bereits existierenden Artefakten vornehmen. Diese können mitunter komplexe Operationen sein, für die die grader instantiation engine ihre Anweisungen über grader-spezifische Elemente in den advanced-Parametern erhält. Dass ein ganzes Artefakt dort enthalten ist, ist zwar nicht verboten, wird vom Anwendungsfall her aber vermutlich eher ein Spezialfall sein. Ein realitätsnäheres Beispiel ist im elektronischen Anhang enthalten[5]. Dort werden zunächst in den Zeilen 13 - 18 einige (fiktive) Standard-Parameter für die grader instantiation engine gesetzt, die diese für ihre Instanziierungen benötigt. Anschließend wird in den Zeilen 20 - 23 der Platzhalter-Parameter und das Encoding für die durch die grader instantiation engine instanziierten Artefakte abgeändert. Ab Zeile 32 wird schließlich eine advanced ArtifactAction für einen Test definiert, die abhängig von dem Wert eines Variationspunktes eine Test-Konfiguration manipuliert. Dies setzt natürlich alles voraus, dass die entsprechende grader instantiation engine dieses Format korrekt interpretieren kann. Eine andere GIE würde möglicherweise anders vorgehen und zum Beispiel alle Parameter immer nur in dem advanced-Parameter einer ArtifactAction zusammen mit dem auszuführenden Befehl erwarten und die Standard-Parameter gar nicht benutzen.

[5]Datei /doc/xml/examples/InstantiationInstructionsExample_Everything.xml

```
[...]
<vpt:instantiation_instructions>
  <vpt:task_action id="advanced_root" action_type="advanced">
    <vpt:action_parameter xsi:type="vpt:advanced_parameter">

      <p:external-resource id="er1" reference="someServer/foo.
         ↪ html">
        <p:description>Sample description</p:description>
        <foo:fooBar attr2="Java">
          <foo:baz>
            <foo:biz a="b"/>
          </foo:baz>
        </foo:fooBar>
      </p:external-resource>

      <p:model-solution id="ms1" comment="Sample␣comment">
        <p:filerefs>
          <p:fileref refid="id000"/>
        </p:filerefs>
      </p:model-solution>

      <p:test id="test1">
        <p:title>SampleTitle</p:title>
        <p:test-type>Sample type</p:test-type>
        <p:test-configuration>
          <p:filerefs>
            <p:fileref refid="id000"/>
          </p:filerefs>
        </p:test-configuration>
      </p:test>

    </vpt:action_parameter>
  </vpt:task_action>
</vpt:instantiation_instructions>
[...]
```

Listing 6.4: Beispiel-XML-Instanzdokument der InstantiationInstructions für den grader instantiation engine stub

6.7.3 Alternative Einbindung von grader instantiation engines

Das in Abschnitt 6.5 vorgestellte Konzept könnte auch auf das momentan verwendete Konzept für die Anbindung der grader instantiation engines übertragen werden. Anstelle, dass die konkrete grader instantiation engine Plugin-Klasse in der Konfigurationsdatei des Instanziierungsservices fest gesetzt wird, könnte hier ebenfalls über Annotations

gearbeitet und somit mehrere grader instantiation engines unterstützt werden, wie es schon in Abschnitt 5.10 diskutiert wurde. Weiterhin könnte das Plugin-Interface derart abgeändert werden, dass es die gesamte Aufgabenschablone als Parameter erhält und diese beliebig verändern kann, um den Plugins freie Hand in ihrer Instanziierung zu lassen. Wie in Abschnitt 6.7.1 erläutert, ist das momentan nicht möglich. Würden die grader instantiation engines aber lokal als Plugin ausgeführt werden, würde das kein Problem mehr darstellen.

6.8 Erweiterung für grader-spezifische Wertemengen

Wie in Kapitel 3.6 angesprochen, sollte das Behandeln grader-spezifischer Wertemengen nur entworfen, nicht aber implementiert werden. Bei der Implementierung des Instanziierungsservices wurde aber darauf geachtet, dass eine entsprechende Erweiterung ohne großen Aufwand möglich ist. Für grader-spezifische Wertemengen gibt es zwei unterschiedliche Anwendungszenarien: 1. Das Abrufen grader-spezifischer Wertemengen und 2. Das Übergeben der Aufgabenschablone und der Wertemenge an die grader instantiation engine, damit diese die Wertebelegung ggf. noch einmal abändern kann.
Um diese Erweiterungen zu implementieren, können dem GraderInstantiationEngine-Plugin entsprechende Methoden hinzugefügt werden, sodass grader instantiation engines diese Funktionalitäten unterstützen können. Für den ersten Anwendungsfall kann einfach eine weitere API-Schnittstelle hinzugefügt werden, die zum Beispiel über einen GET-Request an „/graderspecificvalues“ angesprochen werden könnte. Die Methode, die diese Anfrage verarbeitet, kann die Anfrage einfach an die entsprechende Methode der grader instantiation engine weiterleiten und deren Antwort wiederum an den Aufrufer zurückliefern.
Im zweiten Anwendungsfall kann der InstantiationManager, bevor er mit der Instanziierung beginnt, prüfen, ob grader-spezifische Wertemengen in der Aufgabenschablone vorhanden sind und dann ggf. ebenfalls die grader instantiation engine aufrufen. Die anschließende Instanziierung kann unverändert weiter genutzt werden, sodass der eigentlichen Instanziierung letztlich nur ein weiterer Schritt vorgeschaltet wird.

6.9 Client-Test-Stub

Der Client-Test-Stub dient dazu, den Instanziierungsservice zu testen. Er liegt als ausführbare JAR-Datei vor, die zum Beispiel über die Kommandozeile bedient werden kann. Um ihn möglichst flexibel verwenden zu können, sind alle relevanten Parameter als Kommandozeilenparameter einstellbar. Im Einzelnen sind das die URL, über die der Instanziierungsservice zu erreichen ist, der Benutzername und das Passwort, die zur Authentifizierung genutzt werden sollen und der Pfad zur Aufgabenschablone, die gesendet werden soll. Optional können noch angegeben werden, wie oft der Request

gesendet werden soll, um eine Art „Stress-Test“ zu ermöglichen, sowie der Pfad zur Ausgabedatei, in die die generierte Aufgabeninstanz geschrieben werden soll. Sollte mehr als ein Request gesendet werden, wird nur die Antwort auf den ersten Request auf die Festplatte geschrieben, alle anderen Antworten werden verworfen. Wird der Stub falsch aufgerufen, wird die korrekte Verwendung inkl. aller möglichen Parameter ausgegeben. Soll eine Aufgabeninstanz erzeugt werden, lädt der Stub die Aufgabenschablone, generiert eine zufällige Wertebelegung und sendet beide mit der Anfrage an den Instanziierungsservice.

7 Ergebnisse und Bewertung

7.1 Durchspielen der Anforderungen

Nachfolgend werden die in Kapitel 3 erstellten Anforderungen durchgegangen und überprüft, ob der Instanziierungsservice diese erfüllt.
In Kapitel 3.2 werden die Abläufe aus Nutzer-Sicht beschrieben. Beim ersten Aufrufen einer Aufgabe, soll eine Aufgabeninstanz generiert werden können, allerdings unterschiedlich für Dozenten und Studierende. Studierende sollten eine rein zufällige Instanz erhalten und Dozenten eine bestimmte Wertebelegung mit übergeben können. Wie in Abschnitt 5.1 diskutiert, liegt das Generieren einer Wertebelegung nicht in der Verantwortung des Instanziierungsservices, sondern des Aufrufers. Somit können beide Anforderungen durch dieselbe Schnittstelle abgedeckt werden - für Studierende kann eine zufällige Wertebelegung an den IS übergeben werden und für Lehrpersonen eine selbst zusammengestellte. Weiterhin wird die Anforderung gestellt, dass für den Dialog, über den sich Lehrpersonen eine Wertebelegung konfigurieren können, eine Möglichkeit existieren soll, grader-spezifische Wertemengen abzurufen. Entsprechend den in Abschnitt 3.6 festgelegten Zielen, soll dies aber nur entworfen und nicht implementiert werden - dass und wie eine solche nachträgliche Implementierung möglich ist, wurde in Abschnitt 6.8 vorgestellt. Gleiches gilt für das in Abschnitt 3.3.6 beschriebene Ersetzen grader-spezifischer Wertemengen während des Instanziierungsprozesses.
Ebenfalls in Abschnitt 3.3.6 beschrieben, wird die Möglichkeit, das IS-Grader-Backend-Plugin aufzurufen, das die Instanziierung entweder selbst ausführt oder nur als Adapter für den eigentlichen Grader-Service fungiert. Hierfür gibt es das in Abschnitt 6.7 beschriebene GraderInstantiationEnginePlugin-Interface. Diese Schnittstelle wurde so entworfen, dass die Grader-Instanziierung sowohl lokal, als auch auf einem anderen Server durchgeführt werden kann.
In Abschnitt 3.4 wurde festgelegt, dass der Instanziierungsservice mit unterschiedlichen Formen der Artefaktschablonen-Formulierung umgehen können soll. Genannt wurden einfache Artefaktschablonen, die Platzhalter beinhalten, gültige Artefakte, denen man ihren Schablonen-Charakter nicht ansieht und ungültige Artefakte, denen u.U. Attribute oder Elemente fehlen. Platzhalter-Schablonen werden vom Instanziierungsservice unterstützt, allerdings nur für nicht-numerische Attribute und solche, die keine Referenzen auf IDs beinhalten oder einen festen Wertebereich haben, die entsprechende Erklärung befindet sich in Abschnitt 6.4. Gültige Artefakte werden ohne Einschränkungen unterstützt, ungültige, wie in Abschnitt 6.2.2 beschrieben, allerdings nicht. Dies könnte ggf. aber leicht geändert werden.

Im darauffolgenden Abschnitt 3.5 wird die Integration des Instanziierungsservices an der Hochschule Hannover beschrieben. Dies soll zunächst mit dem Grader Graja und dem LMS Moodle geschehen. Der Instanziierungsservice wurde nicht im Hinblick auf ein bestimmtes Backend-Plugin oder einen bestimmten Nutzer entwickelt, sodass eine Integration dieser beiden Komponenten ohne Probleme möglich ist. Gleichzeitig gewährleistet dies auch die Übertragbarkeit auf andere Systemlandschaften außerhalb der HsH. In Abschnitt 3.6 werden noch einmal grob die Ziele der Arbeit genannt, auf die teilweise bereits eingegangen wurde. Zusätzlich ist dort die Erstellung eines IS-Backend-Plugins für den Grader Graja genannt. Da die Graja Template-Engine zur Zeit noch nicht kompatibel mit dem Instanziierungsservice ist, wurde diese Anforderung im Laufe der Bearbeitung dahingehend abgeändert, dass ein Stub erstellt wird, mithilfe dessen unter möglichst realistischen Bedingungen die Anbindung des Backend-Plugins getestet werden kann. Die Realisierung dieses Stubs wird in Abschnitt 6.7.2 beschrieben. Weiterhin sollte ein einfaches Konsolenprogramm als Client-Stub erstellt werden, dem eine ProFormA-Aufgabenschablone übergeben wird und der sich mit dem IS verbindet, um aus dieser Aufgabenschablone eine Instanz erstellen zu lassen. Ein solches Programm mit vielen Einstellungsmöglichkeiten wurde implementiert und in Abschnitt 6.9 näher erläutert. Die letzte dort genannte Anforderung ist die, vorliegenden Grappa-Code möglichst wiederzuverwenden, sodass im Idealfall eine Bibliothek entsteht, die beide Systeme nutzen können. Diese Anforderung stellte sich bei genauerer Analyse des Grappa-Codes als nicht einfach umsetzbar heraus, da Grappa und der Instanziierungsservice funktional zwar ähnlich sind, aber in Bezug auf die nicht-funktionalen Eigenschaften unterschiedliche Entwurfsziele verfolgen. Diese Anforderung wurde in Absprache mit Robert Garmann und Peter Fricke somit als weniger relevant eingestuft, da vermutlich viel Zeit in Anpassungen und Refactoring für Dinge geflossen wäre, die zwar in Grappa enthalten, für den Instanziierungsservice momentan aber eigentlich nicht notwendig wären. Um der Anforderung dennoch möglichst gerecht zu werden, wurden funktionale Designentscheidungen von Grappa beeinflusst, sodass kleine Teile des Grappa-Codes auch im IS wiederverwendet werden konnten.

7.2 Testfälle

Im nachfolgenden findet sich eine Tabelle der Testfälle, inkl. des erwarteten Ergebnisses, gegen die der Instanziierungsservice erfolgreich getestet wurde. Die Tests wurden unter Ubuntu 16.04 mit Apache Tomcat 9.0.8 und Java 1.8.0_162 durchgeführt.

Nummer	**Beschreibung**	**Erwartung/Ergebnis**
Grundvoraussetzungen		
1.1	Keine task.xml-Datei in der Aufgabenschablone enthalten	IS gibt Fehler zurück

1.2	Keine GradingHints in der task.xml-Datei	IS gibt Fehler zurück
1.3	Keine VariabilityInfo in der task.xml-Datei	IS gibt Fehler zurück
1.4	Keine InstantiationInstructions in der task.xml-Datei	IS gibt Fehler zurück
1.5	InstantiationInstructions sind leer	IS führt Instanziierung durch, aber instanziiert nichts
1.6	Ungültige task.xml-Datei	IS gibt Fehler zurück
1.7	Ungültige InstantiationInstructions	IS gibt Fehler zurück
Grundinstanziierung		
2.1	Instanziieren eines beliebigen Artefaktes nur durch default-Parameter	Erfolgreiche Instanziierung
2.2	Instanziieren eines beliebigen Artefaktes nur durch Artefakt-Parameter	Erfolgreiche Instanziierung
2.3	Instanziieren eines beliebigen Artefaktes mit die default-Parametern überschreibenden Artefakt-Parametern	Erfolgreiche Instanziierung mit Artefakt-Parametern
2.4	Instanziierungsversuch ohne Platzhalter-Parameter	IS gibt Fehler zurück
2.5	Instanziierungsversuch eines embedded file ohne encoding Parameter	Erfolgreiche Instanziierung
2.6	Instanziierungsversuch eines „file“ file ohne encoding Parameter	IS gibt Fehler zurück
2.7	Instanziieerungsversuch eines „file“ file mit ungültiger Codierung	IS gibt Fehler zurück
2.8	Instanziieren eines einzelnen Artefaktes, obwohl mehrere Artefakte Platzhalter enthalten	Nur das angegebene Artefakt wird instanziiert
2.9	Mehrfaches/Rekursives Auflösen von Variationspunkten (z.B. %vp{%vp{a}} zu %vp{b} zu c)	Rekursives Auflösen funktioniert
2.10	Escape-Zeichen der Template-Engine testen (z.B. \%vp{name} wird nicht ersetzt)	Ersetzung wird nicht ausgeführt
Instanziierung aller Artefakttypen		
3.1.a	description	Erfolgreiche Instanziierung
3.1.b	submission-restriction	Erfolgreiche Instanziierung
3.1.c.I	„file“ file	Erfolgreiche Instanziierung

3.1.c.II	embedded file	Erfolgreiche Instanziierung
3.1.d	external-resource	Erfolgreiche Instanziierung
3.1.e	model-solution	Erfolgreiche Instanziierung
3.1.f	test	Erfolgreiche Instanziierung
3.1.g	grading-hints	Erfolgreiche Instanziierung
3.1.h	meta-data	Erfolgreiche Instanziierung
3.2	Falsche Artefakt-Referenz übergeben	IS gibt Fehler zurück
3.3	Fehlende Artefakt-Referenz übergeben	IS gibt Fehler zurück
3.4	Fehlender Artefakt-Typ	IS gibt Fehler zurück
3.5	Falscher Artefakt-Typ	IS gibt Fehler zurück
Grader Instantiation Engine		
4.1.	Instanziieren/Ersetzen von Artefakten durch GIE	
4.1.a	description	Erfolgreiche Instanziierung
4.1.b	submission-restriction	Erfolgreiche Instanziierung
4.1.c.I	„file“ file	Erfolgreiche Instanziierung
4.1.c.II	embedded file	Erfolgreiche Instanziierung
4.1.d	external-resource	Erfolgreiche Instanziierung
4.1.e	model-solution	Erfolgreiche Instanziierung
4.1.f	test	Erfolgreiche Instanziierung
4.1.g	grading-hints	Erfolgreiche Instanziierung
4.1.h	meta-data	Erfolgreiche Instanziierung
4.2.	Hinzufügen von Artefakten durch GIE	
4.2.a.I	„file“ file	Erfolgreiche Instanziierung
4.2.a.II	embedded file	Erfolgreiche Instanziierung
4.2.b	external-resource	Erfolgreiche Instanziierung
4.2.c	model-solution	Erfolgreiche Instanziierung
4.2.d	test	Erfolgreiche Instanziierung
Performance		
5.1.	Zehn mal eine Anfrage senden und Mittelwert bilden (Dauer beim Aufrufer gemessen)	
5.1.a	Mit leeren InstantiationInstructions	256 ms
5.1.b	Mit sehr vielen InstantiationInstructions ohne GIE	273 ms
5.1.c	Mit sehr vielen InstantiationInstructions mit GIE	275 ms
5.2.	Zehn mal n gleichzeitige Anfragen schicken und Mittelwert bilden (mit der task.zip-Datei aus 5.1.c)(Dauer beim Aufrufer gemessen)	
5.2.a	50 gleichzeitige Anfragen	1320 ms
5.2.b	100 gleichzeitige Anfragen	1968 ms
5.2.c	1000 gleichzeitige Anfragen	14193 ms

Deployment		
6.1	IS als .war-Datei manuell außerhalb der IDE in Tomcat deployen	Funktioniert

Tabelle 7.1: Tabelle aller Testfälle

7.3 Bewertung

Wie in Abschnitt 7.1 dargestellt, konnten alle im Laufe der Arbeit herausgearbeiteten Anforderungen letztlich erfüllt werden. Einige anfänglich gestellte Anforderungen wurden nach genauerer Analyse und Rücksprache abgeändert, da eine Umformulierung sinnvoller erschien. Zusätzlich zur Erfüllung aller Anforderungen, zeigen die Performance-Tests, dass auch diese zufriedenstellend ist. Wenngleich die absoluten Zahlenwerte natürlich nicht repräsentativ sind, da sie stark vom unterliegenden System abhängen, so werden die Ergebnisse relativ zueinander gesehen vermutlich auf allen Systemen ähnlich aussehen. Insbesondere der Vergleich der Testergebnisse der Tests 5.1.b und 5.1.c zeigt, dass die Einbindung einer grader instantiation engine seitens des Instanziierungsservices sehr performant ist und es fast ausschließlich darauf ankommt, was genau die grader instantiation engine selbst macht. Eine Implementierung der optionalen grader-spezifischen Wertemengen wäre noch wünschenswert gewesen, wäre in der gegebenen Zeit inklusive der dazu notwendigen Tests etc. allerdings nur schwer zu bewerkstelligen gewesen. Die notwendigen Voraussetzungen dafür sind allerdings gegeben und es sollte leicht möglich sein, dieses nachträglich zu implementieren. Da dieses Feature für die momentan existierenden variablen Programmieraufgaben an der Hochschule Hannover aber auch noch nicht benötigt wird, stehen ersten Feldversuchen des in dieser Arbeit entstandenen Instanziierungsservices nichts mehr im Wege, sobald die verwendeten Nachbarsysteme angepasst sind. Gleiches gilt aber auch für die Verwendung andernorts, da der IS wie bereits angesprochen „nachbarsystem-unabhängig“ entwickelt wurde.

8 Zusammenfassung

In dieser Arbeit wurde zunächst das ProFormaA-Aufgabenformat vorgestellt, das ein hochschulübergreifendes Austauschformat für (Programmier)aufgaben darstellt. Anschließend wurde eine von Prof. Dr. Robert Garmann entwickelte Erweiterung dieses Formats eingeführt, mithilfe deren den Aufgaben Variabilität verliehen werden kann und die die Grundlage für die in dieser Arbeit erstellten Komponenten bildet. Es wurde erläutert, wie ein Instanziierungsservice, der variable ProFormA-Aufgaben entgegen nimmt und daraus ProFormA-kompatible Aufgabeninstanzen generiert in den Hochschulalltag integriert werden könnte und welche Systeme welche Verantwortungen übernehmen könnten. Ausgehend von den daraus entstandenen Anforderungen wurden unterschiedliche Lösungsmöglichkeiten für Teile des Instanziierungsservices diskutiert. Ein besonderes Augenmerk wurde darauf gelegt, welche Aufgaben dem Instanziierungsservice und welche dem Grader zufallen, wie sie erkennen, was sie instanziieren sollen und wie sie zusammenarbeiten. Im Rahmen dessen wurde ein weiteres, „InstantiationInstructions"genanntes, Format entwickelt, das allen diesbezüglich gestellten Anforderungen genügt. So können z.B. alle Artefakte einzeln mit für sie spezifischen Parametern zur Instanziierung angegeben werden, gleichzeitig können zur Vermeidung von Redundanz aber auch in kurzer Form allgemeingültige Parameter und Instanziierungsbefehle für die ganze Aufgabenschablone festgelegt werden. Der gewählte Entwurf des Instanziierungsservices übernimmt das einfache Suchen-und-Ersetzen von Platzhaltern, ist allerdings auch leicht erweiterbar für weitere Instanziierungsmöglichkeiten. Für grader-spezifische oder allgemein komplexere Instanziierungen kann ein grader instantiation engine plugin angebunden werden, das seinerseits Artefakte auf beliebige Weise instanziieren kann. Neben dem eigentlichen Instanziierungsservice ist noch ein grader instantiation engine stub, sowie ein kommandozeilenbasierter Client-Stub entstanden. Der grader instantiation engine stub dient hauptsächlich dem Testen, der Client-Stub nimmt eine Aufgabenschablone entgegen, sendet eine Anfrage an einen Instanziierungsservice und speichert die generierte Aufgabeninstanz ab. Die erstellten Komponenten erfüllen alle Anforderungen und die beobachtete Performance des Instanziierungsservices ist ebenfalls zufriedenstellend. Somit steht ersten Feldversuchen mit variablen Programmieraufgaben - abgesehen von noch nicht angepassten Nachbarsystemen - nichts mehr im Wege.

9 Ausblick

Der implementierte Instanziierungsservice ist noch relativ einfach gehalten, da es sich um eine Art Prototyp handelt. Dies bedeutet gleichzeitig auch, dass es noch viele Verbesserungs- und Erweiterungsmöglichkeiten gibt. Denkbar wären zum Beispiel folgende Dinge:

- Die gleichzeitige Anbindung beliebig vieler grader instantiation engine plugins
- Die Verbesserung einiger nicht-funktionaler Eigenschaften, wie z.B. der Ausfallsicherheit oder der Sicherheit
- Die Nutzung unterschiedlicher Template-Engines und deren eigener Template-Sprachen (vgl. Abschnitt 6.4)
- Das Umstellen von grader instantiation engine plugins auf allgemeine Plugins, wie es in Abschnitt 6.7.3 beschrieben wurde. Gleichzeitig könnten so auch weitere Instanziierungsmöglichkeiten implementiert werden
- Eine nähere Untersuchung des im letzten Absatz von Abschnitt 5.4 beschriebenen Konzepts. Insbesondere, ob ein solcher Entwurf auf längere Sicht einfacher für Lehrpersonen ist
- Eine Einschätzung, wie relevant die in Abschnitt 6.4 genannten Einschränkungen bezüglich der Verwendung von Platzhaltern in der Praxis tatsächlich sind und darauf basierend eine Entscheidung darüber, ob eine Validierung der Aufgabenschablone an der Eingangsschnittstelle tatsächlich sinnvoll ist oder darauf verzichtet werden sollte
- Eine Erweiterung der InstantiationInstructions dahingehend, dass Artefakte auch gelöscht werden können, falls sich die Lösung des Nur-Hinzufügens in der Praxis als unpraktisch erweist
- Ein verbesserter Entwurf der InstantiationInstructions bzgl. Redundanz, wie er im folgenden Kapitel 9.1 beschrieben wird

9.1 Redundanzfreie InstantiationInstructions

Bei Tests mit weiteren Beispielen ist aufgefallen, dass Redundanz entsteht, sobald dieselbe Action für mehrere Artefakte durchgeführt werden soll. In diesen Fällen müssen die ArtifactActions inklusive ihrer Parameter nämlich in jedem TemplateArtifact redundant angegeben werden. Um dieses Problem in Zukunft beheben zu können, werden nachfolgend zwei Lösungsansätze kurz vorgestellt.

Eine erste Möglichkeit wäre, die Definition der ArtifactActions aus den TemplateArtifacts herauszuziehen, sodass diese „global“ für alle TemplateArtifacts verfügbar wären und in diesen referenziert werden könnten. Ein Beispiel solch abgewandelter InstantiationInstructions ist in Listing 9.1 aufgeführt. In diesem Beispiel soll ein in einem file-Artefakt (f4) enthaltenes Skript auf mehreren File-Artefakten (f1, f2, f3) ausgeführt werden und diese instanziieren. Die entsprechende Action wird nur einmal definiert und anschließend mehrfach referenziert.
Der Vorteil dieser Variante ist, dass sie semantisch weiterhin sehr eindeutig ist, indem einzelnen Artefakten bestimmte Instanziierungs-Aktionen zugewiesen werden. Zudem besteht auch eine große Ähnlichkeit zum momentanen Entwurf, sodass keine allzu großen Änderungen notwendig wären.

```
<vpt:instantiation_instructions>
  <vpt:artifact_action id="advanced" action_type="advanced">
       <vpt:action_parameter xsi:type="vpt:advanced_parameter">
            <x:run main-class-file-ref="f4"/>
       </vpt:action_parameter>
  </vpt:artifact_action>

  <vpt:template_artifact id="01" artifact_id="f1" type="file">
       <vpt:artifact_action_ref refid="advanced"/>
  </vpt:template_artifact>

  <vpt:template_artifact id="02" artifact_id="f2" type="file">
       <vpt:artifact_action_ref refid="advanced"/>
  </vpt:template_artifact>

  <vpt:template_artifact id="03" artifact_id="f3" type="file">
       <vpt:artifact_action_ref refid="advanced"/>
  </vpt:template_artifact>
</vpt:instantiation_instructions>
```

Listing 9.1: InstantiationInstructions mit ArtifactAction-Referenzen

Eine weitere Möglichkeit wäre, den Gedanken der Referenzen fortzuführen und zusätzlich darauf zu achten, dass insgesamt wenig Redundanz entsteht. So wird in Listing 9.1 die Referenz auf die ArtifactAction dreimal angegeben. Eine andere Herangehensweise wäre

die Definition von „Blöcken“, die einer Menge von Artefakten eine Menge von Artifact-Actions zuweisen. Ein Beispiel ist in Listing 9.2 aufgeführt. Dort werden zunächst zwei ArtifactActions definiert. Anschließend folgt ein Block („template_artifact_actions“), der zwei Action-Referenzen und drei Artefakt-Referenzen enthält, auf die die referenzierten Aktionen angewendet werden sollen.
Diese Variante ist flexibel und hochgradig redundanzfrei, da sie zusätzlich auf die Mehrfachnennung der Action-Referenzen verzichtet. Sie ist semantisch nicht ganz so klar wie der erste Vorschlag, da sie etwas komplexer ist, ist aber dennoch sehr leicht zu verstehen.

```
<vpt:instantiation_instructions>

  <vpt:artifact_action id="run1" action_type="advanced">
       <vpt:action_parameter xsi:type="vpt:advanced_parameter">
            <x:run main-class-file-ref="f4"/>
       </vpt:action_parameter>
  </vpt:artifact_action>

  <vpt:artifact_action id="run2" action_type="advanced">
       <vpt:action_parameter xsi:type="vpt:advanced_parameter">
            <x:run main-class-file-ref="f5"/>
       </vpt:action_parameter>
  </vpt:artifact_action>


  <vpt:template_artifact_actions id="01">
       <vpt:action_refs>
            <vpt:artifact_action_ref refid="run1"/>
            <vpt:artifact_action_ref refid="run2"/>
       </vpt:action_refs>
       <vpt:artifact_refs>
            <vpt:artifact_ref id="01" artifact_id="f1" type="
               ↪ file"/>
            <vpt:artifact_ref id="02" artifact_id="f2" type="
               ↪ file"/>
            <vpt:artifact_ref id="03" artifact_id="f3" type="
               ↪ file"/>
       </vpt:artifact_refs>
  </vpt:template_artifact_actions>

</vpt:instantiation_instructions>
```

Listing 9.2: InstantiationInstructions mit Referenzen und Blöcken

Tabelle 9.1 listet die vier grundsätzlichen Möglichkeiten der Artefakt-zu-Action-Beziehungen auf (ausgenommen 1:1) und gruppiert diese nach Farben. Die blaue Gruppe ist

	Action A	Act. B	Act. C	Act. D	Act. E	Act. F	Act. G
Artefakt 1	X	X					
Artefakt 2	X	X					
Artefakt 3			X	X			
Artefakt 4					X		
Artefakt 5					X		
Artefakt 6						X	X
Artefakt 7						X	

Tabelle 9.1: Auflistung der möglichen Artefakt-zu-Action-Beziehungen

bereits mit dem aktuellen Entwurf redundanzfrei umsetzbar. Die orange Gruppe spiegelt die Beziehungen in Listing 9.1 wieder, bei der dieselbe Action auf mehrere Artefakte angewendet werden sollen. Dies lässt sich mit beiden Lösungsmöglichkeiten gut umsetzen, wenngleich die zweite Variante wieder auf die Mehrfachnennung der Action-Referenz verzichten würde. Der grüne Fall ist prädestiniert für die zweite Variante und fast genau so in Listing 9.2 aufgeführt, da hier n Aktionen m Artefakten zugeordnet werden. Die gelbe Gruppe kann ebenfalls mit beiden Möglichkeiten umgesetzt werden.

10 Elektronische Anhänge

- / - Root-Folder für das Projekt mit Parent-pom.xml
- grader_instantiation_engine_stub/ - Sub-Projekt-Ordner für den GIES
- instantiation_service - Sub-Projekt-Ordner für den Instanziierungsservice
- is_client_test_stub - Sub-Projekt-Ordner für den Client Test Stub
- README.md - Systemvoraussetzungen & Installationsanweisungen
- doc/bachelorarbeit.pdf - Bachelorarbeit als PDF-Version
- doc/*.tex - Einzelne TEX-Dateien, aus denen die Arbeit besteht
- doc/img/* - In der Arbeit verwendete UML-Diagramme und das Logo der Hochschule
- doc/uml/
  - all_components.asta - Komponentendiagramm mit Abhängigkeiten zwischen LMS, IS, GIE, Grappa & Grader
  - ConceptIS.asta
    - Diagramm AdvancedInstantiationResponse - UML-Klassendiagramm der AdvancedInstantiationResponse
    - Diagramm InstantiationInstructions - UML-Diagramm der Instantiation-Instructions
    - Diagramm IS-Core - UML-Klassendiagramm des Kerns des IS
    - Diagramm MultipleInstancingMechanics - UML-Klassendiagramm des IS mit mehreren InstantiationHandlern
    - Diagramm TransferObject - UML-Klassendiagramm der Transfer-Objekte
    - Diagramm Instantiation-Complete - UML-Sequenzdiagramm des Ablaufs einer Instanziierung inkl. der Erstellung der einzelnen Objekte
    - Diagramm Instantiation-Short - UML-Sequenzdiagramm des Ablaufs einer Instanziierung ohne Erstellung der einzelnen Objekte
  - Grappa_und_Systemumfeld_HS.asta

        * Diagramm Grappa - UML-Klassendiagramm des groben Prinzips von Grappa
        * Diagramm Systemumfeld_HS - Vereinfachte Darstellung des Zusammenhangs von Moogle, Grappa, Graja und aSQLg als UML-Klassendiagramm
    - Interne_Ablaeufe.asta
        * Diagramm Aufgabe_abgeben - UML-Sequenzdiagramm des High-Level-Ablaufs beim Abgeben einer Aufgabe
        * Diagramm Aufgabe_erstellen - UML-Sequenzdiagramm des High-Level-Ablaufs beim Erstellen einer Aufgabe
        * Diagramm Aufgabe_generieren - UML-Sequenzdiagramm des High-Level-Ablaufs beim Generieren einer Aufgabe
        * Diagramm Dozent_Dialog - UML-Sequenzdiagramm des High-Level-Ablaufs beim Erstellen des Auswahldialoges für Lehrpersonen
    - Schnittstellen.asta - Schnittstellen des Instanziierungsservices und des IS-Grader-Backend-Plugins inkl. der grader-spezifischen Wertemengen
- doc/xml/
    - AdvancedInstantiationResponse.xsd - XML-Schema der AdvancedInstantiationResponse
    - InstantiationInstructions.xsd - XML-Schema der InstantiationInstructions
    - taskxml.xsd - XML-Schema des ProFormA-Aufgabenformats
    - examples/
        * AdvancedInstantiationResponseExample.xml - Beispiel-XML-Instanzdokument einer AdvancedInstantiationResponse
        * InstantiationInstructionsExample_AdvancedOnly.xml - Beispiel-XML-Instanzdokument für InstantiationInstructions mit nur advanced TaskActions
        * InstantiationInstructionsExample_Empty.xml - Beispiel-XML-Instanzdokument für leere InstantiationInstructions
        * InstantiationInstructionsExample_Everything.xml - Beispiel-XML-Instanzdokument für InstantiationInstructions mit allen möglichen Elementen
        * InstantiationInstructionsExample_SimpleOnlyShort.xml - Beispiel-XML-Instanzdokument für InstantiationInstructions einer einfachen TaskAction

- InstantiationInstructionsExample_SimpleOnlyWithDefaultParams.xml - Beispiel-XML-Instanzdokument für InstantiationInstructions mit Standard-Parametern und mehreren einfachen Instanziierungen

10.1 Anhang: CD

Glossar

Grader Programm, das eine Einreichung automatisch bewertet. 8, 10–12, 14–19

LMS Lernplattforms bzw. Learning Management Systeme (LMS) sind komplexe Content-Management-Systeme, die der Bereitstellung von Lerninhalten und der Organisation von Lernvorgängen dienen - *Definition Wikipedia vom 27. Februar 2018*. 11, 13–19

Literatur

[1] Oliver J. Bott, Peter Fricke, Uta Priss und Michael Striewe, Hrsg. *Automatisierte Bewertung in der Programmierausbildung*. Waxmann Verlag, 2017. ISBN: 978-3-8309-3606-0.

[2] Robert Garmann. "Der Grader Graja". In: *Automatisierte Bewertung in der Programmierausbildung*. Hrsg. von Oliver J. Bott, Peter Fricke, Uta Priss und Michael Striewe. Waxmann Verlag, 2017, S. 173–189. ISBN: 978-3-8309-3606-0.

[3] Robert Garmann. "Ein Schnittstellen-Datenmodell der Variabilität in automatisch bewerteten Programmieraufgaben". In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018)*. Hrsg. von S. Krusche et. al. CEUR-WS.org. Ulm, Germany, March 06, 2018. URL: `http://ceur-ws.org/Vol-2066/`.

[4] Robert Garmann, Felix Heine und Peter Werner. "Grappa - die Spinne im Netz der Autobewerter und Lernmanagementsysteme". In: *DeLFI 2015: Die 13. e-Leraning Fachtagung Informatik*. Hrsg. von Hans Pongratz und Reinhard Keil. abgerufen am 27. Februar 2018. Bonn: Gesellschaft für Informatik, 2015, S. 169–181. URL: `https://dl.gi.de/bitstream/handle/20.500.12116/2048/169.pdf?sequence=1`.

[5] Felix Heine und Carsten Kleiner. "Der Grader aSQLg". In: *Automatisierte Bewertung in der Programmierausbildung*. Hrsg. von Oliver J. Bott, Peter Fricke, Uta Priss und Michael Striewe. Waxmann Verlag, 2017, S. 191–206. ISBN: 978-3-8309-3606-0.

[6] Sven Strickroth, Michael Striewe, Oliver Müller, Uta Priss, Sebastian Becker, Oliver Rod, Robert Garmann, Oliver J. Bott und Niels Pinkwart. *ProFormA: An XML-based exchange format for programming tasks*. abgerufen am 26. Februar 2018. 2015. URL: `https://eleed.campussource.de/archive/11/4138`.